(51) International Patent Classification[7]: G06F 15/76

(21) International Application Number:
PCT/EP2004/009640

(22) International Filing Date: 30 August 2004 (30.08.2004)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
03019428.6	28 August 2003 (28.08.2003)	EP
03025911.3	5 November 2003 (05.11.2003)	EP
103 57 284.8	5 December 2003 (05.12.2003)	DE
03028953.2	17 December 2003 (17.12.2003)	EP
03079015.8	17 December 2003 (17.12.2003)	EP
04002604.9	5 February 2004 (05.02.2004)	EP
04002719.5	6 February 2004 (06.02.2004)	EP
04003258.3	13 February 2004 (13.02.2004)	EP
04004885.2	2 March 2004 (02.03.2004)	EP
04075654.6	2 March 2004 (02.03.2004)	EP
04005403.3	8 March 2004 (08.03.2004)	EP
04013557.6	9 June 2004 (09.06.2004)	EP
04018267.7	2 August 2004 (02.08.2004)	EP
04077206.3	2 August 2004 (02.08.2004)	EP

(71) Applicant (for all designated States except US): PACT XPP TECHNOLOGIES AG [DE/DE]; Muthmannstrasse 1, 80939 München (DE).

(72) Inventors; and
(75) Inventors/Applicants (for US only): VORBACH, Martin [DE/DE]; Gotthardstrasse 117A, 80689 München (DE). THOMAS, Alexander [DE/DE]; c/o PACT XPP Technologies AG, Muthmannstrasse 1, 80939 München (DE).

(74) Agent: PIETRUK, Claus, Peter; European Patent Attorney, Heinrich-Lilienfein-Weg 5, 76229 Karlsruhe (DE).

(81) Designated States (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States (unless otherwise indicated, for every kind of regional protection available): ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

(54) Title: DATA PROCESSING DEVICE AND METHOD
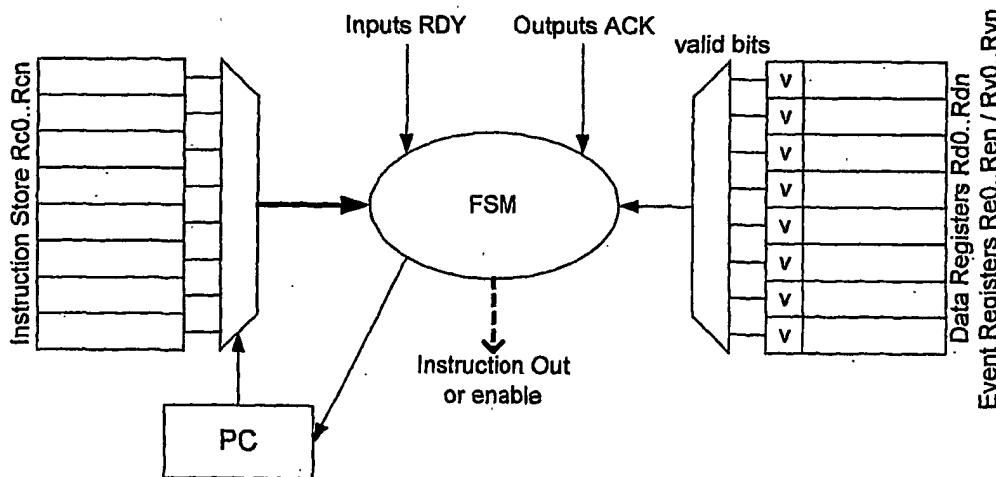
(57) Abstract: A data processing device comprising a multidimensional array of coarse grained logic elements processing data and operating at a first clock rate and communicating with one another and/or other elements via busses and/or communication lines operated at a second clock rate is disclosed, wherein the first clock rate is higher than the second and wherein the coarse grained logic elements comprise storage means for storing data needed to be processed.

WO 2005/045692 A2

Data processing device and method

The present invention relates to reconfigurable computing. In
particular, the present invention relates to improvements in
the architecture of reconfigurable devices.

Reconfigurable data procesing arrays are known in the art. Re-
ference is being made to the previous applications and/or pub-
lications of the present applicant/assignee all of which are
encorporated herein by way of reference. Accordingly, the de-
vices described hereinafter may be multidimensional (n>1) ar-
rays comprinsing coarse grained computing and/or data opera-
tion elements allowing for runtime reconfiguration of the en-
tire array or parts thereof, preferably in response to a sig-
nal indicating reconfigurability to a loading unit (CT, CM or
the like).

Now, several of these data procesing arrays have been built
(i.e. Xppl, XPP128, XPP2, XPP64). It is however desirable to
improve the known device further as well as to improve methods
of its operation.

Accordingly, in order to achieve this object there will be de-
scribed a number of improvements allowing separately or in
commmom to improve the performance and /or power consumption
and /or cost of the device.

A first way to improve the known devices is to improve the
functionability of each single processor element. It has been
previously suggested to include a ring-memory (RINGSPEICHER)
in the array, to store instructions in the ring-memory and to
provide a pointer that points to one of the ring-memory adres-
ses so as to select an instruction to be carried out next.
Furthermore, it has been suggested to provide at least one
„shadow configuration" and to switch over between several con-
figurations /shadow configurations. Anotrher or additional
suggestions has been designated as „wave reconfiguration".

While these known methods improve the performance of a recon-
figurable device, there seems to be both a need and a possi-
bility for further improvements.

It is to be understood that while in the following descrip-
tion, a detailed example is given, for example with respect to
the number of registers given associated with eachj PAE, it is
not deemed necessary to provide an ALU with exactly this num-
ber of registers. Rather, it will be understood by the average
skilled person that deviations from the explicitly described
embodiment are easily feasible and that the detailed level of
description stems from an effort to provide an examplary PAE
and not from the wish to restrict the scope of invention.-

1 Overview of changes vs. XPP XPP-II

1.1 ALU-PAE Architecture

In the suggested improved architecture, a PAE might e.g. comprise 4 input ports and 4 output ports. Embedded with each PAE is the FREG path newly named DF with its dataflow capabilities, like *MERGE, SWAP, DEMUX* as well as *ELUT*.

2 input ports Ri0 and Ri1 are directly connected to the ALU. Two output ports receive the ALU results.
Ri2 and Ri3 are typically fed to the DF path which output is Ro2 and Ro3.
Alternatively Ri2 and Ri3 can serve as inputs for the ALU as well. This extension is needed to provide a suitable amount of ALU inputs if *Function Folding* (as described later) is used. In this mode Ro2 and Ro3 serve as additional outputs.

Associated to each data register (Ri or Ro) is an event port (Ei or Eo).

It is possible, albeit not necessary to implement an additional data and event bypass BRi0-1, BEi0-. The decision depends on how often Function Folding will be used and how many inputs and outputs are required in average.
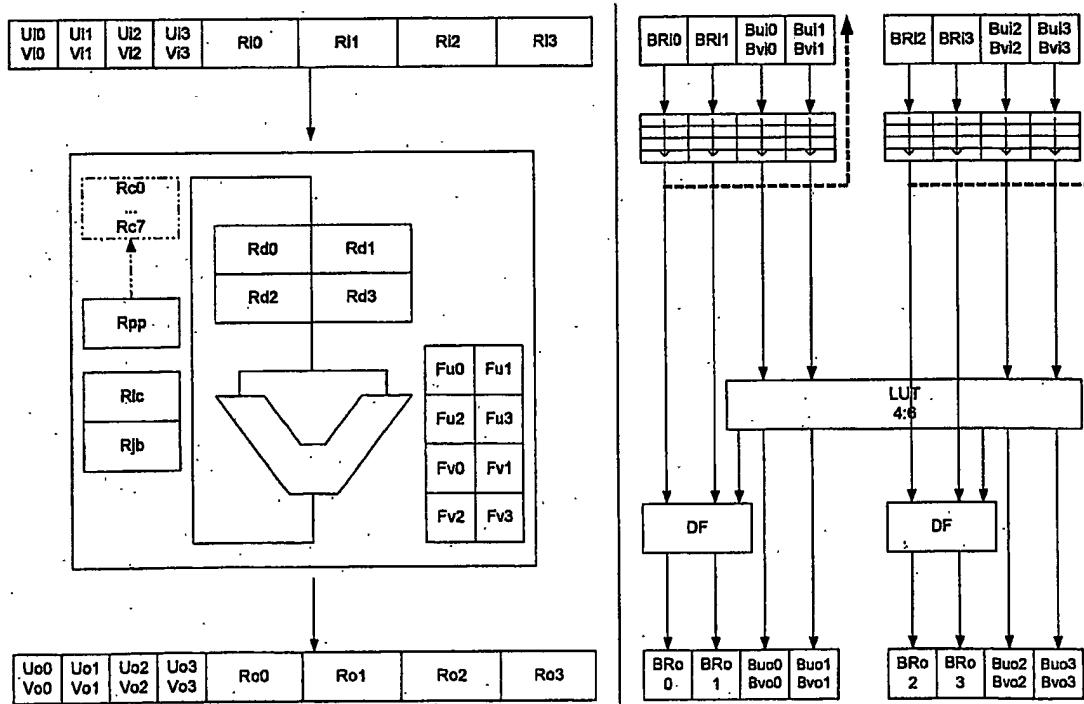
1.1.1 Other extensions

SIMD operation is implemented in the ALUs to support 8 and 16 bit wide data words for i.e. graphics and imaging.

Saturation is supported for ADD/SUB/MUL instructions for i.e. voice, video and imaging algorithms.

1.2 Function Folding

1.2.1 Basics and input/output paradigms

Within this chapter the basic operation paradigms of the XPP architecture are repeated for a better understanding based on Petri-Nets. In addition the Petri-Nets will be enhanced for a better understanding of the subsequently described changes of the current XPP architecture.

In most arrays each PAE operates as a data flow node as defined by Perti-Nets. (Some arrays might have parts that have other functions and should thus be not considered as a standard PAE). A Petri-Net supports a calculation of multiple inputs and produces one single output. Special for a Perti-Net is, that the operation is delayed until all input data is available.

For the XPP technology this means:
   1. all necessary data is available
   2. all necessary events are available
The quantity of data and events is defined by the data and control flow, the availability is displayed at runtime by the handshake protocol RDY/ACK.
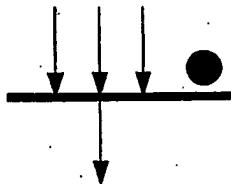
The thick arbor indicates the operation, the dot on the right side indicates that the operation is delayed until all inputs are available.

Enhancing the basic methodology function folding supports multiple operations – maybe even sequential – instead of one, defined as a *Cycle*. It is important that the basics of Petri-Nets remain unchanged.
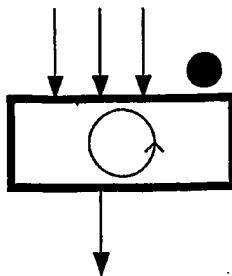
Typical PAE-like Petri-Nets consume one input packet per one operation. For sequential operation multiple reads of the same input packet are supported. However, the interface model again keeps unchanged.

Data duplication occurs in the output path of the Petri-Net, which does not influence the operation basics again.
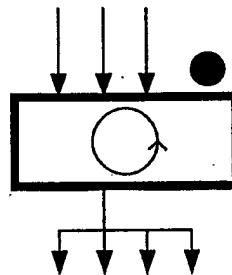
1.2.2 Method of Function Folding

One of the most important extensions is the capability to fold multiple PAE functions onto on PAE and execute them in a sequential manner. It is important to understand that the intention is not to support sequential processing or even microcontroller capabilities at all. The intention of Function Folding is just to take multiple dataflow operations and map them on a single PAE, using a register structure instead of a network between each function.

One goal may be to save silicon area by rising to clock frequency locally in the PAEs. An additional expectation is to save power since the busses operate at a fraction of the clock frequencies of the PAEs. Data transfers over the busses, which consume much power, are reduced.

Just one PAE (not drawn to scale)
PAE Frequency 200MHz
Bus Frequency 50MHz

XPP V2 Rcunction
PAE Frequency 50MHz
Bus Frequency 50MHz

I/O allocation
Register allocation

STEP 1   STEP 2   STEP 3a   STEP 3b
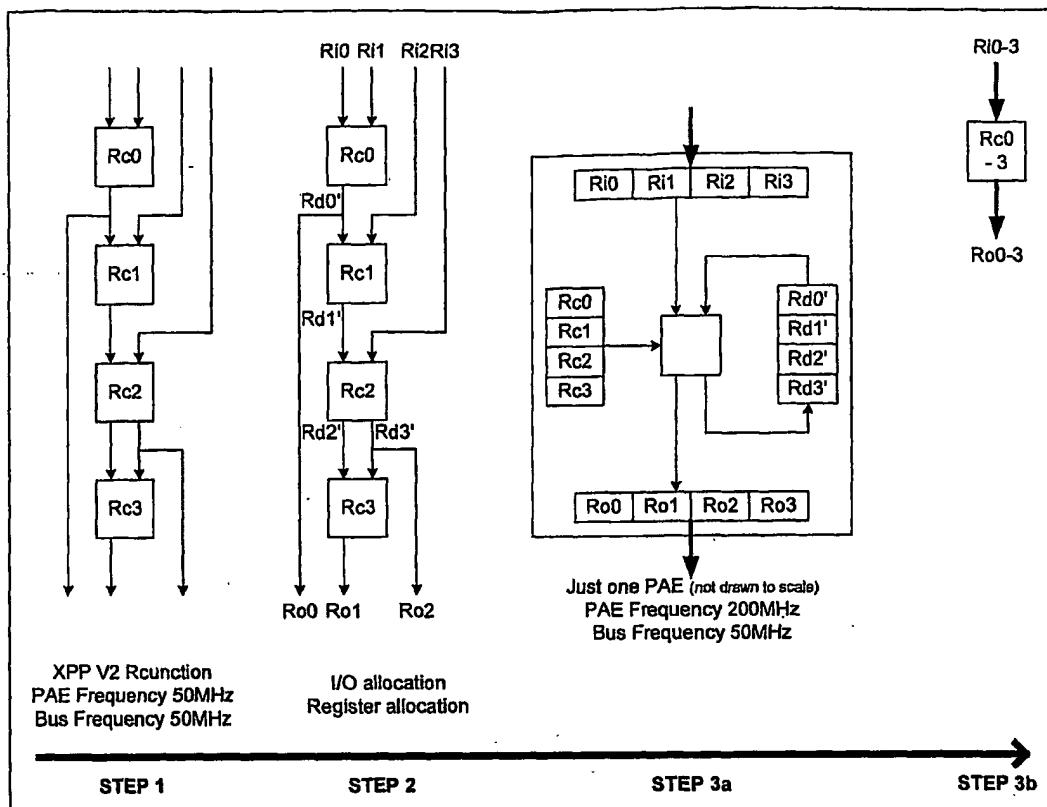
· The internal registers can be implemented in different ways,
e.g. in one of the following two:

1. dataflow model
Each register (r') has a valid bit which is set as soon as
data has-been written into the register and reset after the
data has been read. Data cannot be written if valid is set,
data can not be read if valid is not set. This approach imple-
ments a 100% compatible dataflow behaviour.

2. sequencer model
The registers have no associated valid bits. The PAE operates
as a sequencer, whereas at the edges of the PAE (the bus con-
nects) the paradigm is changed to the XPP-like dataflow behav-
iour.

Even if at first the dataflow model seems preferable, it has
major down sides. One is that a high amount of register is
needed to implement each data path and data duplication is
quite complicated and not efficient. Another is that sometimes
a limited sequential operation simplifies programming and
hardware effort.
Therefore it is assumed consecutively that sequencer model is
implemented. Since pure dataflow can be folded using automatic
tools the programmer should stay within the dataflow paradigm
and not be confused with the additional capabilities. Auto-

matic tools must take care i.e. while register allocation that the paradigm is not violated.

The following figure shows that using sequencer model only 2 registers (instead of 4) are required:

Just one PAE (not drawn to scale)
PAE frequency 200MHz
Bus frequency 50MHz

XPP V2 function
PAE frequency 50MHz
Bus frequency 50MHz

I/O allocation
Register allocation

STEP 1   STEP 2   STEP 3a   STEP 3b
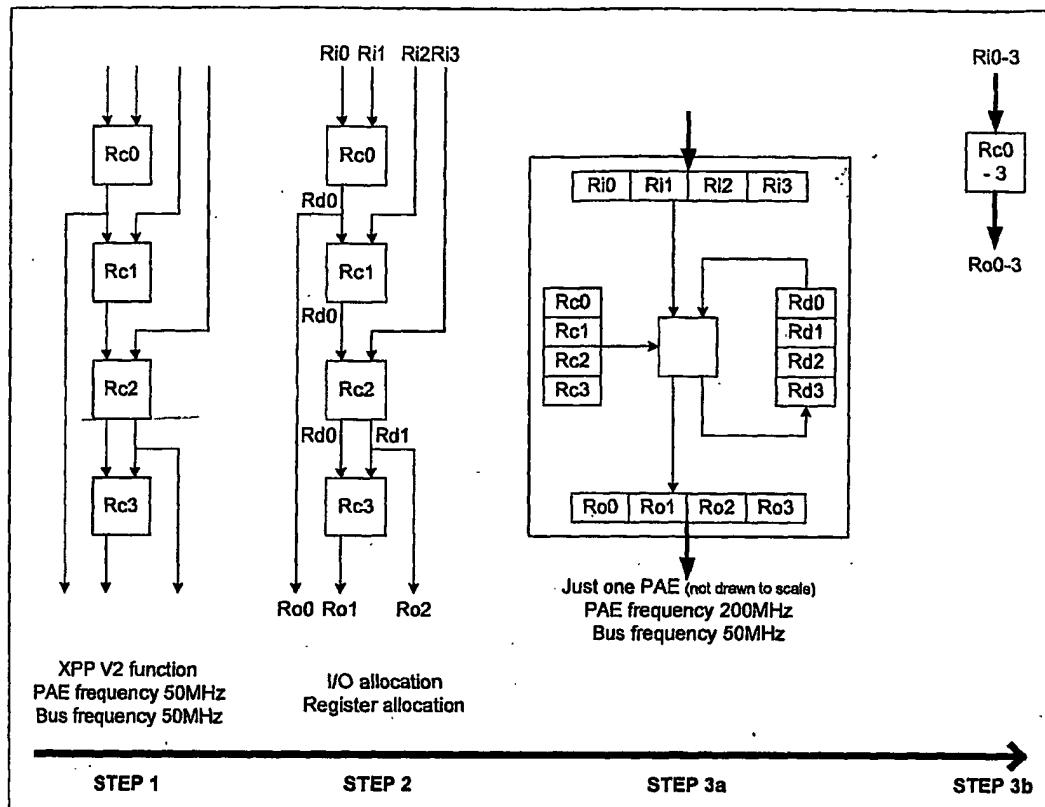
For allowing complex function like i.e. address generation as well as algorithms like "IMEC"-like data stream operations the PAE has not only 4 instruction registers implemented but 8, whereas the maximum bus-clock vs. PAE-clock ration is limited to a factor of 4 for usual function folding.

It is expected that the size of the new PAE supporting Function Folding will increase by max. 25%. On the other hand 4 PAEs are reduced to 1.

Assuming that in average not the optimum but only about 3 functions can be folded onto a single PAE a XPP64 could be replaced by a XPP21. Taking the larger PAEs into account the functionality of a XPP64 XPP-II should be executable on a XPP XPP-III with an area of less than half.

The function folding method and apparatus as well as otrher
further improvements will be described in even more detailed
hereinafter.

Equality of internal data registers and bus transfers

The function fold concept realises two different models of
data processing:

a) Sequential model, wherein within the PAE the same rules ap-
   ply as in von-Neuman- and Harvard-processors.
b) PACT VPU-model, wherein data are calculated or operated upon
   in arbitrary order according to the PETRI-Net-Model (data
   flow + synchronisation).

Due to the unpredictability of the arrival of data at the in-
put registers (IR) a deadlock or at a least significant reduc-
tion in performance could occur if the commands in RC0...RCn
were to be performed in a linear manner. In particular, if
feed-backs of the PAE outputs to the inputs of the PAE are
present, deadlocks might occur. This can be avoided if the in-
structions are not to be processed in a given order but rather
according to the possibility of their processing, that is, one
instruction can be carried out as soon as all conditions of
the VPU-model are fulfilled. Therefore, for example, once all
RDY-handshakes of incoming data, ACK-handshakes of outgoing
data and, if necessary, triggers (including their handshakes)
are valid, then the instruction can be carried out. As the FF
PAE has data additionally stored in internal registers, their
validity and status has to be checkable as well in a preferred
embodiment. Therefor, every internal data register (RD0...RDn)
is separately assigned a valid bit indicating whether or not
valid data are present in the register. When writing data into
the register, valid is set, when reading, valid is reset. Data

can be read only if "valid" is set and can be written only if
"valid" is not set. Accordingly, the valid flag corresponds
most closely to the status that is produced in the state ma-
chines of bus systems by the transmittal of RDY/ACK-
handshakes. It is a preferred embodiment and considered to be
inventive to provide a register with a status bit in that way.

It is therefore possible to carry out instructions at the time
when all conditions for the execution – again very similar to
PETRI-nets are fulfilled.

Basically, there are two methods available for selection of
instruction and control of their execution described herein
after.

Method A: FF PAE Program Pointer
(Finite State Machine & Program Pointer-Approach)
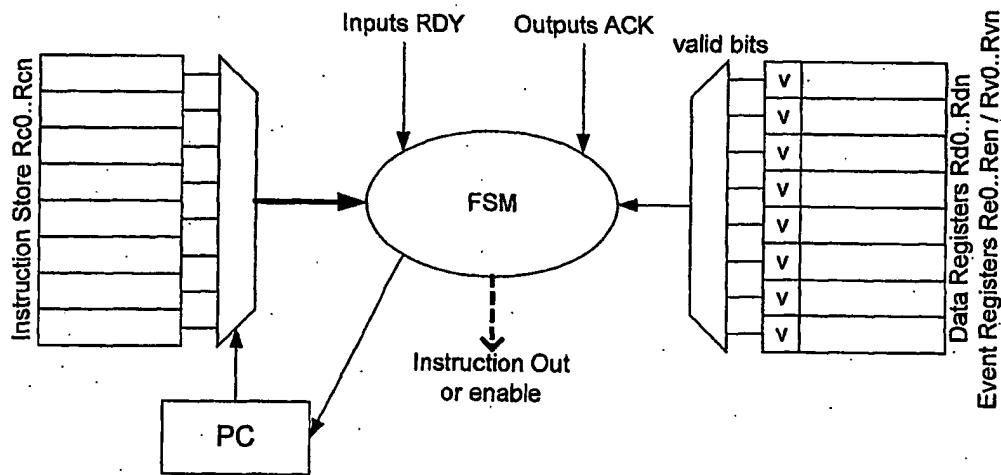
Fig. 1

According to the control principle of sequential processors, a
program counter is used to select a certain instruction within
the instruction memory. A finite state machine controls the
program counter. This finite state machine now checks whether
or not all conditions for the instruction in RC (PC), that is

the instruction, onto which the PC (Program Counter) points, are fulfilled. To do so, the respective RDY- and/or ACK-handshakes of the in- and/or outputs needed for the execution of the instructions are checked. Furthermore, the valid-flags of the internal registers to be read (RD0..RDn) are checked so as to control whether or not they are set, and the valid-flags of those internal registers (RD0..RDn) into which is to be written, are checked whether they are not set. If one of the conditions is not fulfilled, the instructions will not be carried out. PC is controlled to count further, the instruction is skipped and the next instruction is selected and checked as described.

The advantage of this method is the compatibility with sequential processor models. The disadvantage resides in the necessity to test and to skip instructions. Both of which might result in significant losses of performance under certain circumstances.

Method B: FF PAE Program Pointer
(Enabler & Arbiter-Approach)

Inputs RDY   Outputs ACK

Instruction Out

Instruction Store Rc0..Rcn

em
em
em
em
em
em
em
em

Testlogic

valid bits

v
v
v
v
v
v
v
v

Data Registers Rd0..Rdn
Event Registers Re0..Ren / Rv0..Rvn

ISel
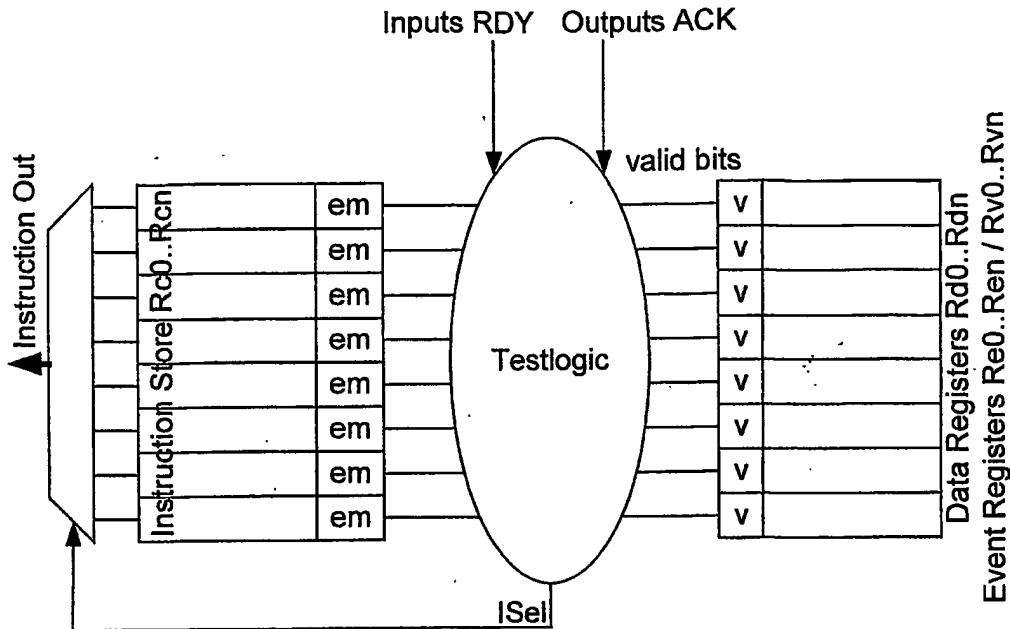
Fig. 2

This method is based upon the possibility to test all instruc-
tions in Rc0..Rcn in parallel. In order to save the expense of
the complete decoding of array instructions, each RC is as-
signed an entry in an evaluation mask field, the length of
which corresponds to the maximum number of states to be
tested; therefore, for every possible RDY- or ACK-trigger-
signal (as well the RDY/ACKs of the triggers) as well as for
every valid bit in RD0...RDn two bits are available indicating
whether or not the respective signal is to be set or not set;
or, whether the state of the signal is unimportant for the
execution of the instruction.

Example mask

InData-RDY		OutData-ACK		InTrigger			OutTrigger-ACK		Rd Data Valid	
Rdy value	don't care	Ack value	don't care	trigger value	rdy value	don't care	ack value	don't care	valid value	don't care

The mask shows only some entries. At In-Trigger, both the
state of the trigger (set, not set) as well as the value of
the trigger (trigger value) can be tested via RDY-value.

A test logic testing via for example the Line Control de-
scribed herein after all instructions in parallel. Using an
arbiter, an instruction of the set of all executables is se-
lected. The arbiter controls the instruction multiplexer via
ISel according to the transferral of the selected instructions
to the PAE.

The Line Control has one single line of Boolean test logic for
every single instruction. By means of an ExOR-gate (e) the
value of the signal to be tested against the setting in em of
the line is checked. By means of an OR-gate (+) respectively,
a selection is carried out, whether the checked signal is
relevant (don't care). The results of all checked signals are
ANDed. A logic 1 at the output of the AND-gates (&) shows an
executable instruction. For every RC, a different test-line
exists. All test-lines are evaluated in parallel. An arbiter
having one of a number of possible implementations such as a
priority arbiter, Round-Robin-Arbiter and so forth, selects
one instruction for execution out of all executable instruc-
tions. There are further implementations possible obvious to
the average skilled person. Those variants might be widely
equivalent in the way of operation and function. In particu-
lar, the possibility of using "negative logic" is to be men-
tioned.
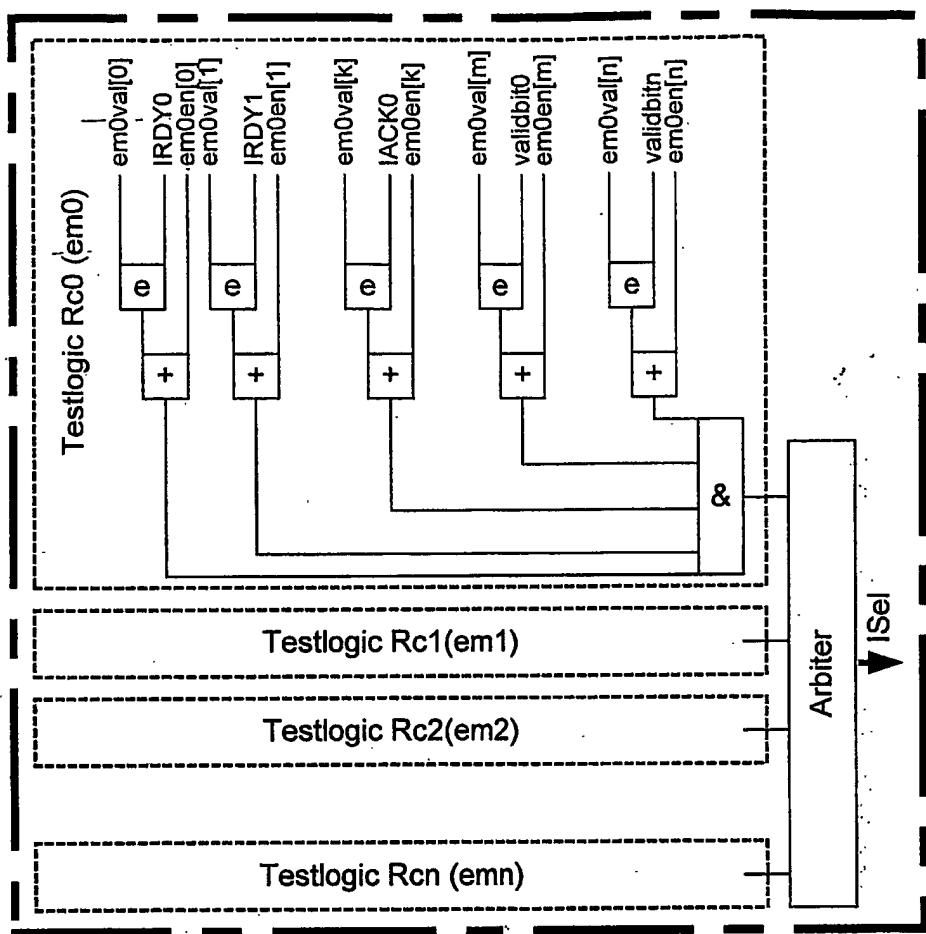
Fig. 3

The following figure gives an overview of the entire cir-
cuitry:
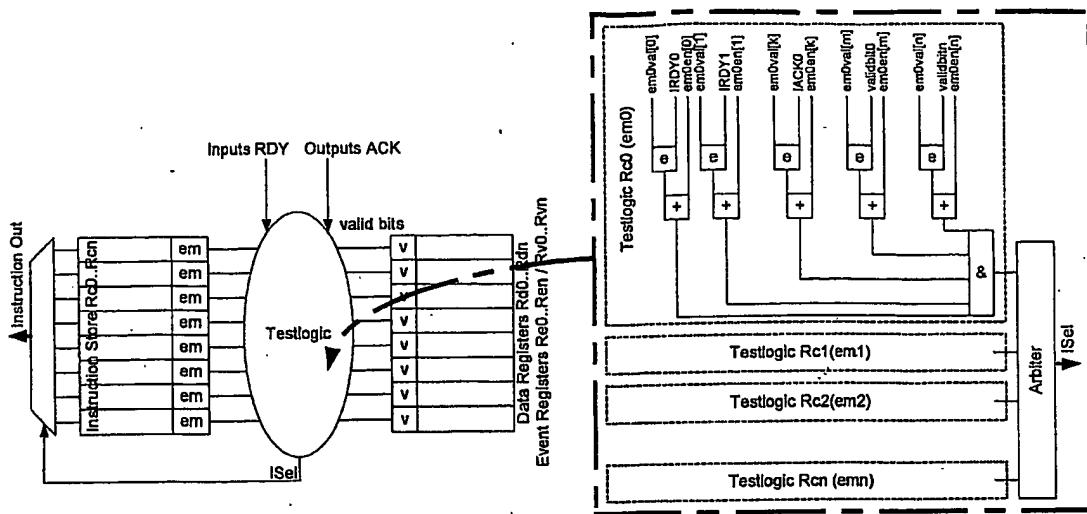
Fig. 4

Advantages of the method are:

- Significantly fast, in view of the fact that one instruc-
  tion can be carried out in every single clock

- Reduced power consumption, since no energy is wasted on
  disgarded cycles which is in particular advantageous to the
  static power dissipation.

- Similar hardware expense as in the sequential solution when
  using small and medium sized configuration memories (RC)
  therefor similar costs.

Disadvantages:

- Likely to be significantly more expensive on large RC;
  therefore, an optimisation is suggested for a given set of
  applications.

- In order to implement the sequencer mode (compare other
  parts of the application) the program counter having an FSM
  must be provided for. The FSM then is restricted to the
  tasks of the sequencer so that the additional expenses and
  the additional costs are relatively low.

Depopulated Busses according to the State of the Art

All busses assigned to a certain PAE are connected to the in-
put registers (IR) or the output registers of the PAE are con-
nected to all busses respectively (compare for example DE 100
50 442.6 or the XPP/VPU-handbooks of the applicant).

It has been realised that PAEs, in particular FF PAEs, allow
for a depopulation of bus interconnects, in particular, if
more IR/OR will be available compared to the State of the Art
of the XPP as previously known. The depopulation, that is the
reductions of the possibilities to connect the IR or ER onto
the busses can be symmetrically or asymmetrically. The depopu-
lation will typically amount to 20 to 70 %. It is significant
that the depopulation will not or not significantly effect the
interconnectability and/or the routability of an algorithm in
a negative way.

The method of depopulation is particularly relevant in view of
the fact that several results can be achieved. The hardware-
expense and thus the costs of the bus systems can be reduced
significantly; the speed of the busses is increased since the
gate delay is reduced by the minimisation of connecting
points; simultaneously, the power consumption of the busses is
reduced.

A preferred depopulation according to the VPU-architecture ac-
cording to the State of the Art, however, with more IR/OR is
shown in the following figure.

optional direct horizontal
next neighbor connect
(4x DEMUX at output (OR),
4x MUX at input (IR))
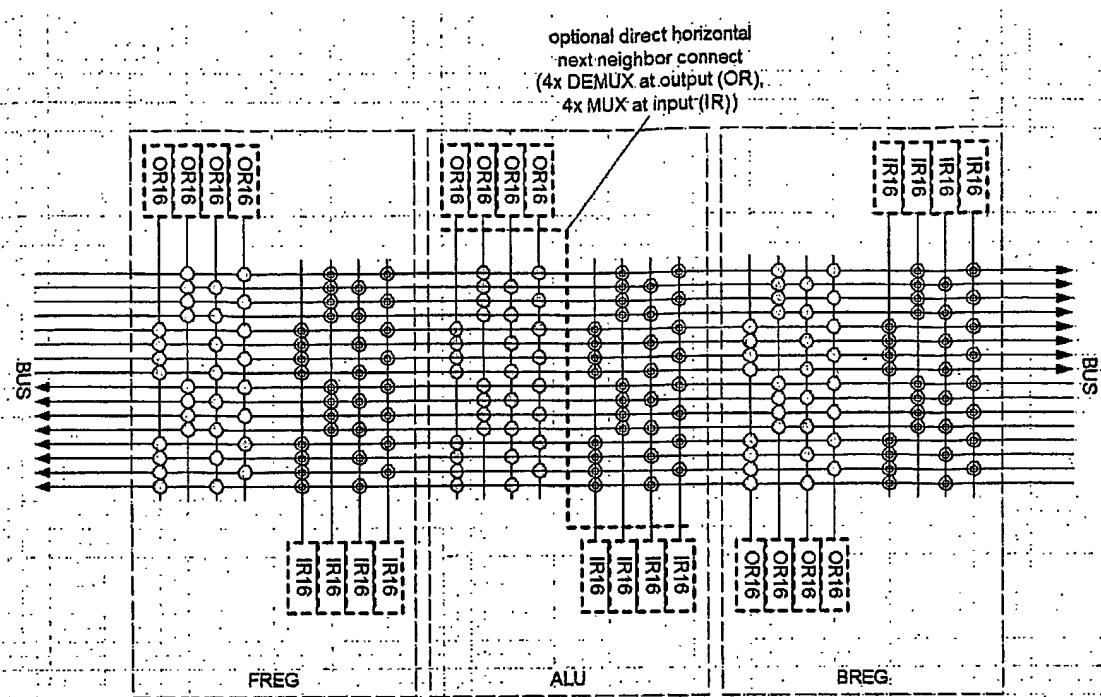
Fig. 5

In particular, reference is being made to an optional exten-
sion of the bus architecture allowing for a direct next neigh-
bour data transfer of two adjacent PAEs, in particular two
PAEs placed one onto the other. Here, the outputs (OR) of one
PAE are directly connected to a dedicated bus which is then
directly connected to the inputs (IR) of a neighbouring PAE
(compare next figure). The figure only shows an horizontal
next neighbour bus, however, in general, vertical busses are
possible as well.

In the figure, the shaded circles stand for possible bus con-
nects: MUX. Double circuits stand for a connection from the
bus: DeMUX.

Changes of the PAE IO

The following figure shows the State of the Art of a PAE implementation as known from XPU128, XPP64A and described in DE 100 50 442.6

The known PAE has a main data flow in the direction from top to bottom to the main ALU in the PAE-core. At the left and right side, data channels are placed additionally transmitting data along the main data flow direction, once the same direction as the main data flow (FREG) and once in the reverse direction (BREG). On both sides of the PAE, data busses are provided that run in the reverse direction of the main data flow of the PAE and onto which the PAE as well as FREG and BREG are connected. The architecture of the State of the Art requires eight data busses for each PAE side as well as four transfer channels for FREG/BREG for typical applications.

Bus = 8
FREG, BREG = 4
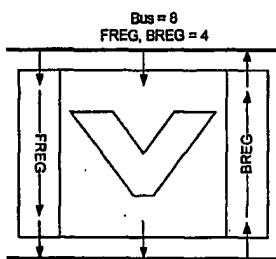
Fig. 6

The bus system of the State of the Art has switching elements, register elements (R), each at the side of the PAEs. The switching elements allow for the disruption of a bus segment or disconnection to a neighbouring bus, the register elements allow the construction of an efficient pipelining by transferring data through the register, so as to allow for higher transferral band-width. The typical latency in vertical direction for next-neighbour-transmitting is 0 per segment, however is 0,5-1 in horizontal direction per segment and higher frequencies.
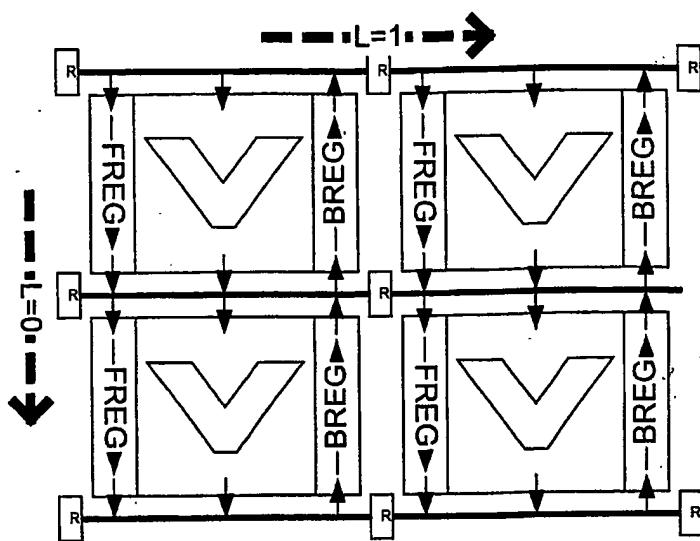
Fig. 7

Now, a modified PAE structure is suggested, wherein two ALUs,
each having a different main data flow direction are provided
in each PAE, allowing for significantly improved routability.
On one hand, the tools used for routing are better and sim-
pler, on the other hand, a significant reduction in hardware
resources is achieved. First tests shows that the number of
busses necessary in horizontal direction is reduced by about
25 % over the State of the Art. The vertical connects in
FREG/BREG (= BYPASS) can even be reduced by about 50 %. Also,
it is no more necessary to distinguish between FREG and BREG
as was necessary in DE 100 50 442.6.

Bus = 6
FREG, BREG = 2
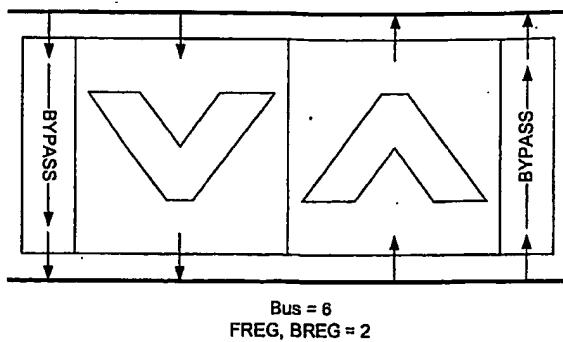
Fig. 8

The double-ALU structure has been further developed to an ALU-PAE having inputs and outputs in both directions. Using automatic routers as well as hand-routed applications, further additional significant improvements of the network topology can be shown. The number of busses necessary seems to be reduced to about 50 % over the State of the Art, the number of vertical connects in the FREG/BREG (= BYPASS) can be reduced by about 75 %.

Bus = 4
FREG, BREG = 1
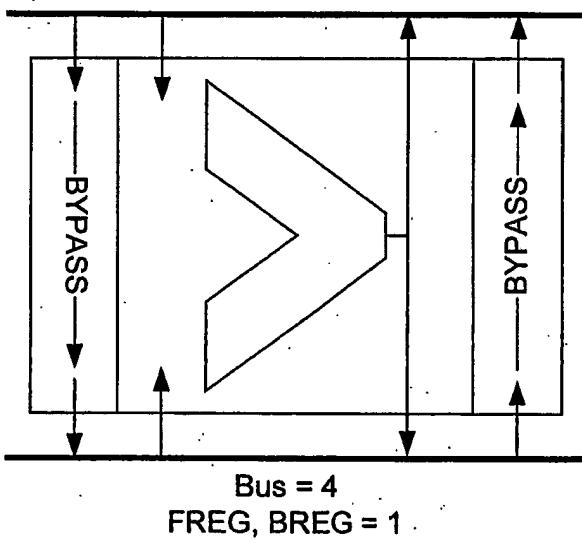
Fig. 9

For this preferred embodiment which can be used for conventional as well as for function fold ALUs, it is possible to place register and switching elements in the busses in the

middle of the PAE instead of at the sides thereof (see Fig.
below).
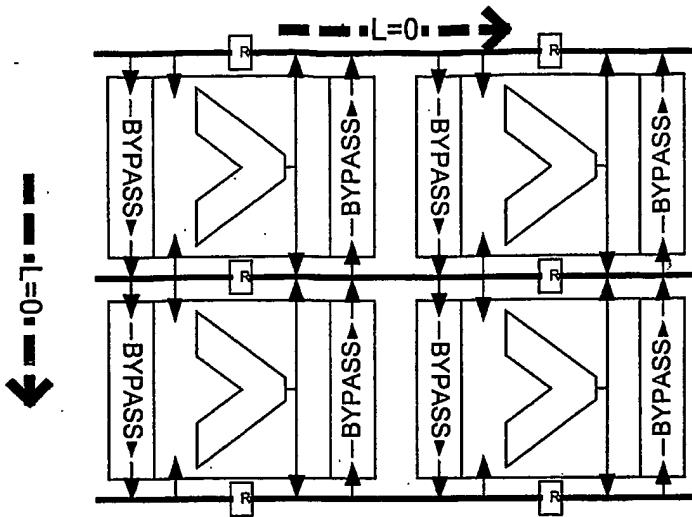
Fig. 10

In this way, it is possible even for high frequencies to
transmit data in horizontal direction to the respective neigh-
bouring PAE without having to go through a register element.
Accordingly, it is possible to set up next neighbour connec-
tions in vertical and horizontal directions which are latency
free (compare State of the Art and drawings referring to de-
populated busses). The example of the interconnections shown
in the respective figure allows transferral having zero la-
tency in vertical direction and horizontally from left to
right. Using an optimisation of PAE interface structure a la-
tency free next neighbouring transmission in both horizontal
directions can be achieved. If in every corner of the PAE in-
put register (IR, arrow of bus into PAE) from bus and output
register (OR, arrow from PAE to bus) to the bus are imple-
mented, each neighbouring PAE can exchange data without la-
tency (see Fig).
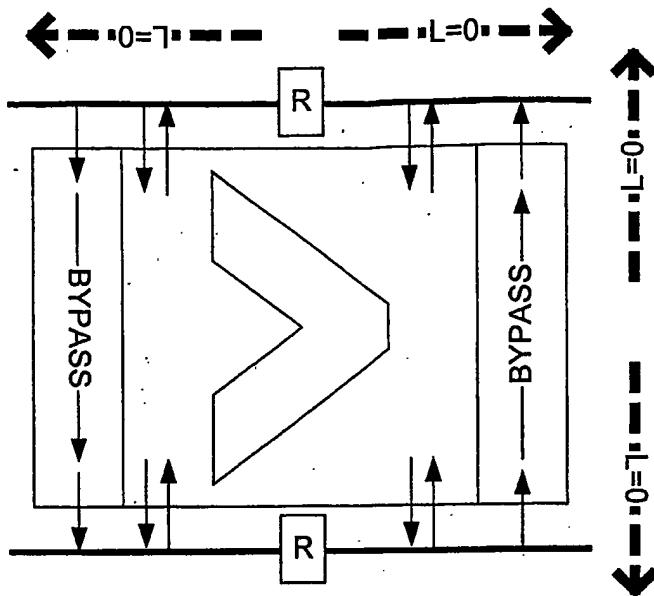
Fig. 11

It is possible to further optimise the above disclosed PAE ar-
rangement. This can be done by using no separate bypass at all
in all or some of the PAEs. The perferred embodiment comprises
two ALUs, one of these being "complete" and having all neces-
sary functions, for example-multiplication and BarrelShift
while the second has a reduced instruction set eliminating
functions that require larger arrays such as multiplication
and BarrelShift. The second ALU is in a way replacing BYPASS
(as drawn). There are several possible positions for the reg-
ister in switching elements per bus system, and two of the
preferred positions per bus are shown in Fig. 12 below in dot-
ted lines.

R   R

+ MULtipl
+ BarrelShFT

BYPASS

- MULtipl
- BarrelShFT

BYPASS

R   R
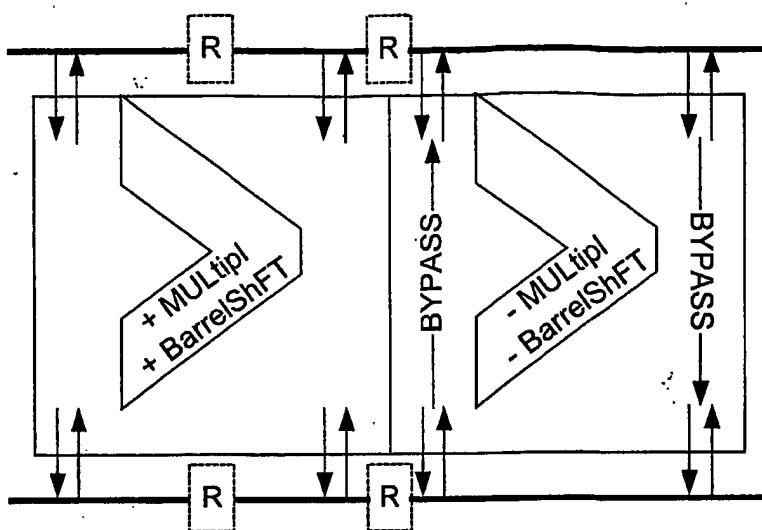
Fig. 12

Both ALUs comprise additional circuits to transfer data be-
tween the busses so as to implement the function of the by-
pass. A number of possible ways of implementations exist and
two of these shall be explained as an example.

a) Multiplexer
Configurable multiplexers within the ALU are connected so that
ALU inputs are bypassing the ALU and are directly connected to
their outputs.

b) MOVE instruction
A MOVE instruction, stored in Rc0..Rcn is transferring within
the respective processing clock of the function fold the data
according to the input specified within the instruction to the
specified output.

Superscalarity/ Pipelining
It is possible and suggested as first way of improving per-
formance to provide roughly superscalar FF ALU-PAEs which cal-

culate for example 2,4,8 operations per bus clock @ FF=2,4,8,
even while using the MUL opcode.

The basic concept is to make use of the VALID-flags of each
internal register. MUL is implemented as one single opcode
which is pipelined over two stages.
MUL takes its operands from the input registers Ri and stores
the results into internal data registers Rd. VALID is set if
data is stored into Rd. ADD (or any other Opcode, such as
BSFT) uses the result in Rd if VALID is set; if not the execu-
tion is skipped according to the specified VALID behaviour.
In addition the timing changes for all OpCodes, if the MUL in-
struction is used inside a PAE configuration. In this case all
usually single cycle OpCodes will change to pipelined 2 cycle
OpCodes. The change is achieved by inserting a bypass able
multiplexer into the data stream as well as into control.

The following program will be explained in detail:
    MUL (Rd0, Rd1), Ri0, Ri1;
    ADD Ro0, Rd1, Ri2;

In the first bus-cycle after configuration (t_0) MUL is executed
(assuming the availability of data at Ri0/1). The register
pair Rd0/1 is invalid during the whole bus-cycle, which means
during both FF-PAE internal clock cycles. Therefore ADD is not
executed in the 2nd clock cycle. After t_0 the result of MUL is
written into the register pair, which VALID flags are set at
the same time.
In t_1 new data is multiplied. Since the VALID is set for Rd0/1
now the ADD command is executed in the 2nd clock cycle, but
takes 2 clock cycles for over all execution. Therefore operand
read and result write is inline for both operations, MUL as
well as ADD.

The result of a MUL-ADD combination is available with 2 clocks
latency in a FF=2 ALU-PAE. For FF >= 6 no latency is inserted.

However since multiplication and all other commands are proc-
essed in parallel the machine streams afterwards without any
additional delays.

FF = 2:

2BT latency

FF = 4:

1BT latency

FF>= 6:

zero latency

wire length: 0.5mm copper
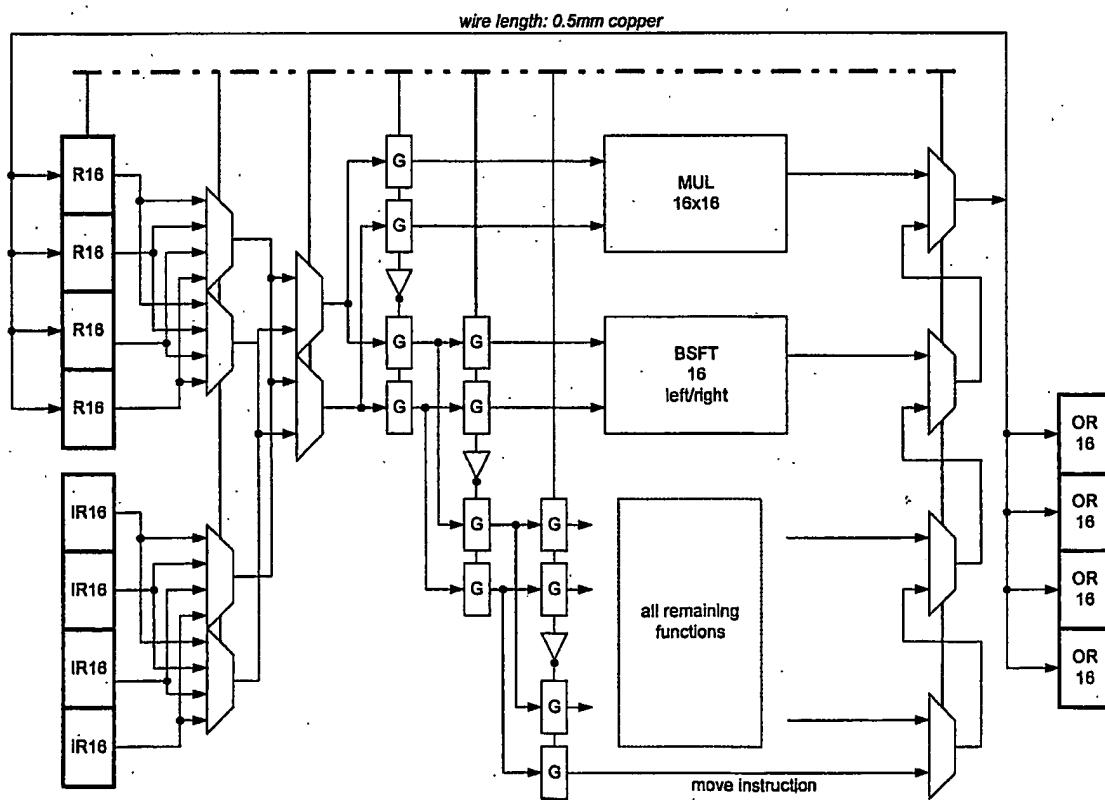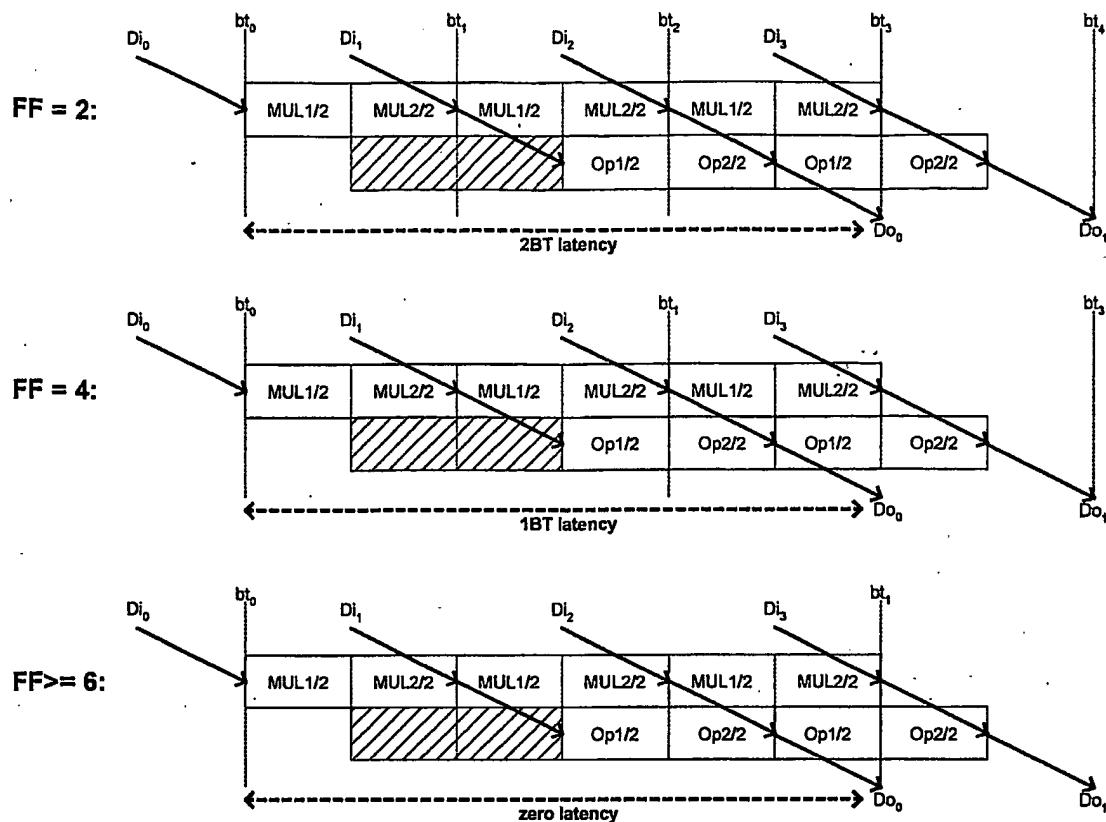
If there are OpCodes besides MUL which require 2 clock cycles for execution (e.g. BSTF) the architecture must be modified to allow at least 3 data writes to registers after the second internal clock cycle.
The data path output multiplexer gets 2 times larger as well as the bus system to the output registers (OR) and the feedback path to the internal data registers (Rd).
If accordingly defined for the OpCodes, more than 4 internal registers can be used without increasing the complexity by using enables (en) to select the specific register to write in the data. Multiple registers are connected to the same bus, e.g. Rd0, Rd4, Rd8, Rd12. However not all combinations of register transfers are possible with this structure. If e.g. MUL uses Rd0 and Rd1 the following registers are blocked for the OpCode executed in parallel: Rd4,5,8,9,12,13.

Register map:

Rd0	Rd4	Rd8	Rd12
Rd1	Rd5	Rd9	Rd13
Rd2	Rd6	Rd10	Rd14
Rd3	Rd7	Rd11	Rd15

Datapath architecture:
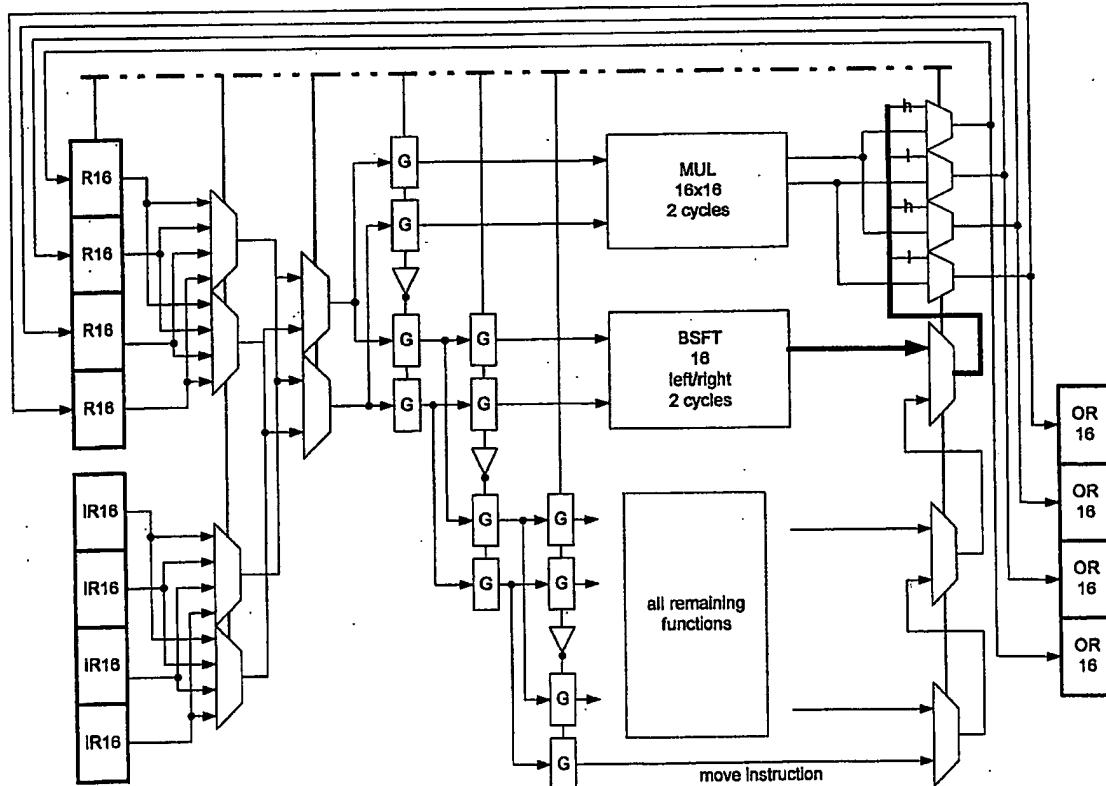
The Sequencer PAEs

Since there is a need to be able to run control flow dominated
applications on the XPP III as well, Sequencer PAEs will be
introduced. Such a PAE can be thought of as a very simple kind
of processor which is capable to run sequential code within
the XPP. This allows the efficient implementation of control
flow oriented applications like the H.264 Codec on the array
whereas with SEQ-PAEs missing the realization would be more
difficult and resource consuming.

The SEQ-PAEs are not build from scratch. Instead such a tile
will be build up by a closely coupling of a ALU-PAE and neigh-
boring RAM-PAE, which can be seen in Figure 1.

RAM-PAE

ALU-PAE   configurable
               Sequencer
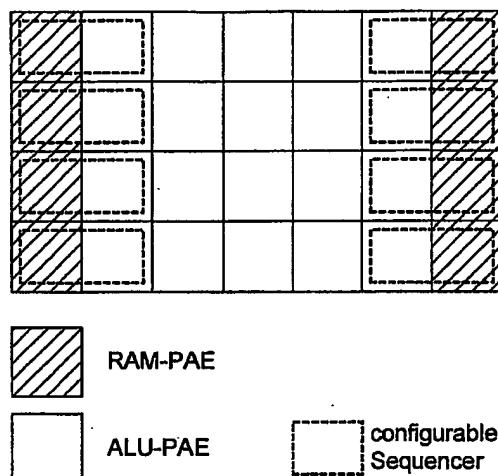
Figure 1 : Configurabel Sequencer

Therefore the functionality of the ALU- as well as RAM-PAE has
to be enhanced to be able to fulfill the requirements of such
a SQE-PAE. This information will be given next.

ALU-PAE Enhancements

The extended version of the ALU-PAE is given in Figure 2. To
the right border the registers which are controlling the dif-
ferent modules can be seen. Those registers will be used in
normal- as well as in SEQ-mode. Therefore the appropriate con-
trol signals from the local configuration manager and the RAM-
PAE are first merged by OR-Gates and then are forwarded to the
register whereas it has to be ensured that in normal mode the
signals from the RAM-PAE are 0 and vice versa.

Further more, since the ALU-PAE marks the execution part of
the tiny processor, there is a need to transfer values to and
from the internal register directly to the RAM. Therefore a
additional multiplexer **AM1** is inserted in the multiplexer hi-
erarch of section 2. In the normal mode this multiplexer feeds

the word from its predecessor to the next stage whereas in the
SEQ mode an immediate value provided by the Imm. Register will
be delivered. In addition in SEQ mode a value of one of the
internal registers can be delivered to the RAM-PAE via the
output of the multiplexer. However, it has also **to be consid-
ered to provide a "LOAD reg, imm" since this is not much slo-
wer than „ADD reg, reg, imm"**

To enable the RAM-PAE to write data to the internal register
of the ALU-PAE another multiplexer is inserted in the multi-
plexer chain of section 4. Similar to the scenario given above
this multiplexer will only be activated in SEQ mode whereas in
normal mode this multiplexer will just forward the data of its
predecessor. In one preferred embodiment, it is suggested to
place **RS2 behind BSFT-Mux in view of the delay. Data could be
written into the internal registers via this. (LOAD reg, imm)]**
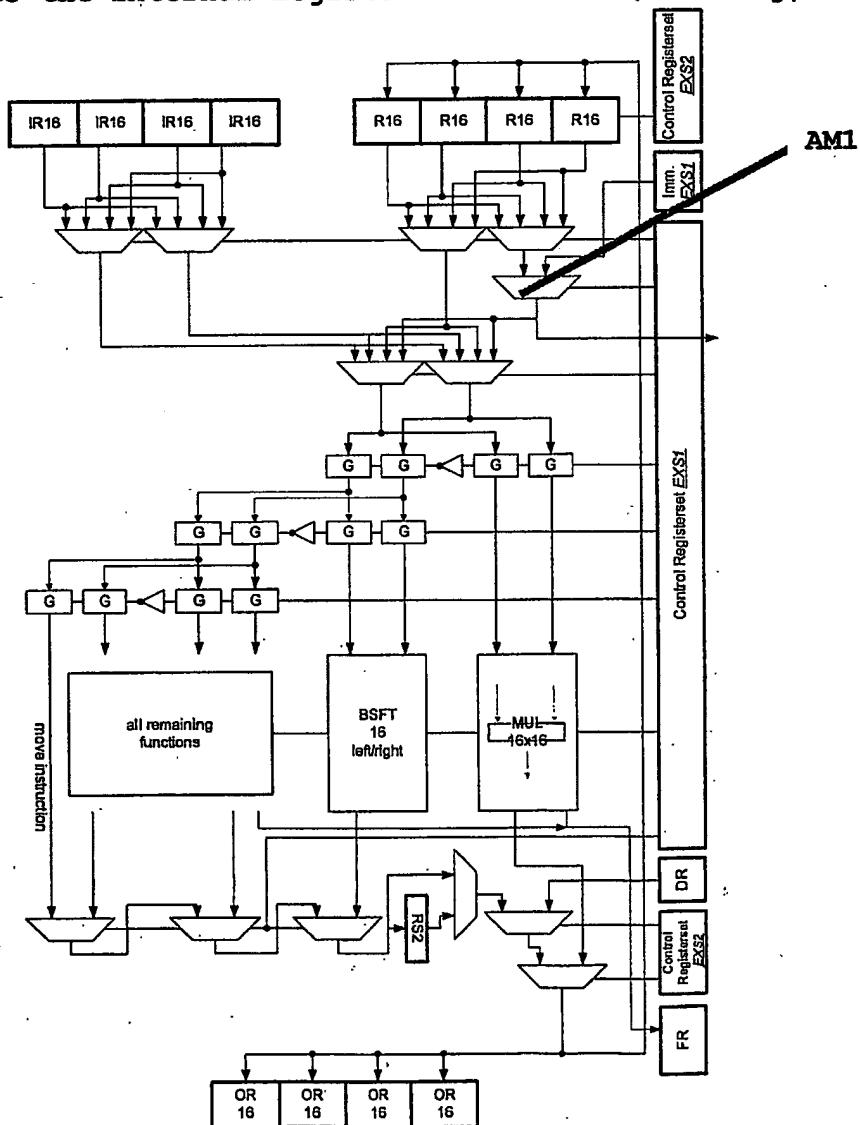
Figure 2 : Enhanced Version of the ALU-PAE

As it has already been discussed, data can be processed during
one or two cycles by the ALU-PAE depending on the selected
arithmetic function. Due to the auto synchronization feature
of the XPP and due to the fact that in normal mode a succes-
sive operation will not start before the previous one is fin-
ished, it does not really care if an operation lasts one or
two clock cycles. Whereas the tile is working in SEQ mode
there is a difference since we assume to have a pipeline char-
acter. This means that a one cycle operation could run in par-
allel with a two cycle module where the operation would be
executed in stage two at this time. Due to the limited multi-
plexing capacities of a word - 16 Bit - only one result could
be written to the connected registers whereas the other one
would be lost. In general there are three possibilities to
solve this problem.

The first one could be that the compiler is capable to handle
this problem. This would mean that it has to know about the
pipeline structure of the whole SEQ-PAE as well as of a tile
in detail. To prohibit a parallel execution the compile would
have to add a NOP to every two cycle instruction for the
structure given above. However this idea seems not to be con-
venient due to the strong relation between the hardware struc-
ture and the compiler. The drawback would be that every time
changes are made to the hardware the compile would most likely
have to be trimmed to the new structure.
The second idea could be to recognize such a situation in the
decode stage of the pipeline. If a two cycle instruction is
directly followed by an instruction accessing a one stage
arithmetic unit it has to be delayed by one clock cycle as
well.
The last possibility is to make the complete ALU-PAE look like
a two stage execution unit. Therefore only one register has to
be included in the multiplexer chain of section four right af-
ter the crossover from the multiplexer separating the one
stage of the two stage modules. Obviously, this is preferred.

Comparing the last to ideas the third one seems to be the best
one since only one register has to be inserted. If we a closer
look to the second solution special logic would be needed for
analyzing the disallowed combination of instructions as well
as logic for stopping the program counter (PC) and the in-
struction retardation. It has to be assumed that this logic
would require much more area than the registers as well as the
fact that the delay of the logic would possibly increase the
critical path.
Since it has to be distinguished between the SEQ and the nor-
mal mode where a one cycle execution should still be avail-
able. This possibility is given by a multiplexer which allows
to bypass the RS2 Register as shown in the corresponding fig-
ure. (Figure 2)

The RAM-PAE

A short description of the stages

To get the SEQ-PAE working there still has to be provided more
functionality. Right now the RAM-PAE will take care of it. As
a first approach for realizing the sequencer a four stage
pipeline has been chosen. The stages are, as it can be seen in
Figure 3:

- The fetch stage
- The decode stage
- The execution stage 1
- The execution stage 2

In the fetch stage the program counter for the next clock cy-
cle will be calculated. This means that it will be either in-
cremented by 1 via a local adder or one of the program count-
ers from the decode or execution stage 2 will be selected. The
program counter of the execution stage thereby provides the
address if a call instruction occurred whereas the program
counter of the execution stage provides the PC if there has
been a conditional jump. Right now the branch address can ei-
ther be calculated out of the current PC and a value which ei-
ther be an immediate value or a value from a internal regis-
ters of the ALU-RAM – indirect addressing mode – or an abso-
lute value. This e.g. is necessary if there is return from a
subroutine to the previous context whereas the according abso-
lute PC will be provided by the stack bank.
In the decode stage the instruction coming from the code bank
will be decoded. Necessary control signals and, if needed, the
immediate value for the internal execution stage 1 as well as
for the execution stage 1 of the ALU-PAE will be generated.
The signals include the control information for the multiplex-
ers and gating stages of section two of the ALU-PAE, the op-
eration selection of the ALU's tiles, e.g. signed or unsigned
multiplication, and the information whether the stack pointer
(SP) should be in/decremented or kept unchanged in the next
stage depending on the fact if the instruction is either a
call or jump. In case a call instruction occurred a new PC
will be calculated in parallel and delivered to the fetch
stage.
Furthermore the read address and read enable signal to the
data bank will be generated in case of a load instruction.
In the execution stage 1, which by the way is the first stage
available on the ALU as well as on the RAM-PAE, the control
signals for execution stage 2 of the ALU-PAE are generated.
Those signal will take care that the correct output of one of
the arithmetical tiles will be selected and written to the en-
abled registers. If the instruction should be a conditional

jump or return the stack pointer will be modified in this
stage. In parallel the actual PC will be saved to the stack
bank at the address give by the Rsp EX1 register in case of a
branch. Otherwise, in case of a return, the read address as
well as the read enable signal will be applied to the stack
bank.
In execution stage 2 the value of the PC will be calculated
and provided to the multiplexer in the fetch stage in case of
a jump. At the time write address and write enable signal to
the data bank are generated if data from the ALU have to be
saved.
Instead of two adders, it is possible to provide only one in
the rpp path.
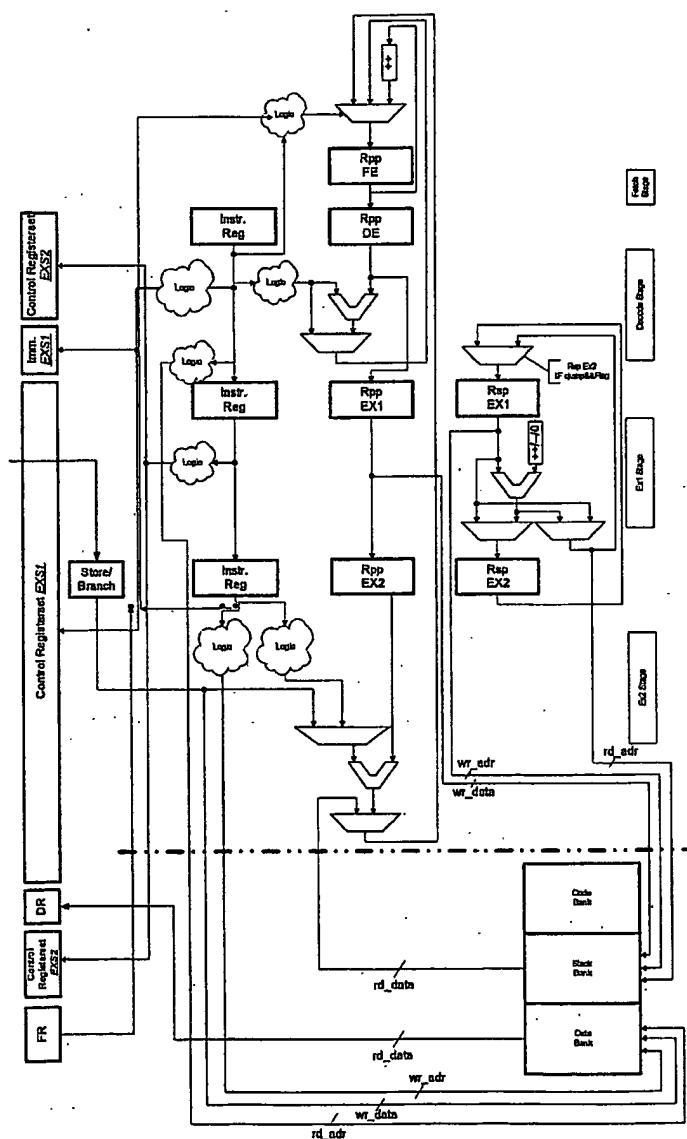
Figure 3 : Overview of the RAM-PAE

Pipeline actions

In the following section a short overview of the actions that are taking place in the four stages will be given for some basic instructions. It should help to understand the behaviour of the pipeline. Since the instruction which is going to be discussed will be available at the instruction register the actins of the fetch stage will be omitted in this representation.

IR: Instruction Register
DR: Data Register
DB: Data Bank
SBR : Store/Branch Register

Instruction: Load value from data bank to R[n]

ALU-PAE	RAM-PAE
decode stage	
	IR_ex1 <- IR_ex2 Control Registerset EXS1 <- 0x0 Imm. *EXS1* <- 0x0 Rpp_ex1 <- Rpp_de DB_radr <- imm
Execution stage 1	
	IR_ex2 <- IR_ex1 Control Registerset EXS2 <- enable R, set mux section 4 Rpp_ex2 <- Rpp_ex1 DR <- DB_radr [imm] Rsp_ex2 <- Rsp_ex1
Executinon stage 2	
R[n] <- DR	

Instruction: Sore value from R[n] to data bank

ALU-PAE	RAM-PAE
decode stage	
	IR_ex1 <- IR_ex2 Control Registerset EXS1 <- enable mux section 2 Imm. *EXS1* <- 0x0 Rpp_ex1 <- Rpp_de
Execution stage 1	
SBR <- R[n]	IR_ex2 <- IR_ex1 Control Registerset EXS2 <- 0x0 Rpp_ex2 <- Rpp_ex1 Rsp_ex2 <- Rsp_ex1
Executinon stage 2	
	DB_wradr <- imm DB_wrdata <- SBR

1.3 Array Structure

First advantages over the prior art are obtained by using
function folding PAEs. These as well as other PAEs can be im-
proved.

The XPP-II structure of the PAEs consumes much area for FREG
and BREG and their associated bus interfaces. In addition feed
backs through the FREGs require the insertion of registers
into the feedback path, which result not only in an increased
latency but also in a negative impact onto the throughput and
performance of the XPP.

A new PAE structure and arrangement is proposed with the ex-
pectation to minimize latency and optimize the bus intercon-
nect structure to achieve an optimized area.

The XPP-III PAE structure does not include BREGs any more. As
a replacement the ALUs are alternating flipped horizontally
which leads to improved placement and routing capabilities es-
pecially for feedback paths i.e. of loops.
Each PAE contains now two ALUs and two BP paths, one from top
to bottom and one flipped from bottom to top.

PAE Structure and   PAE Structure and
arrangement V2.0   arrangement V2.2
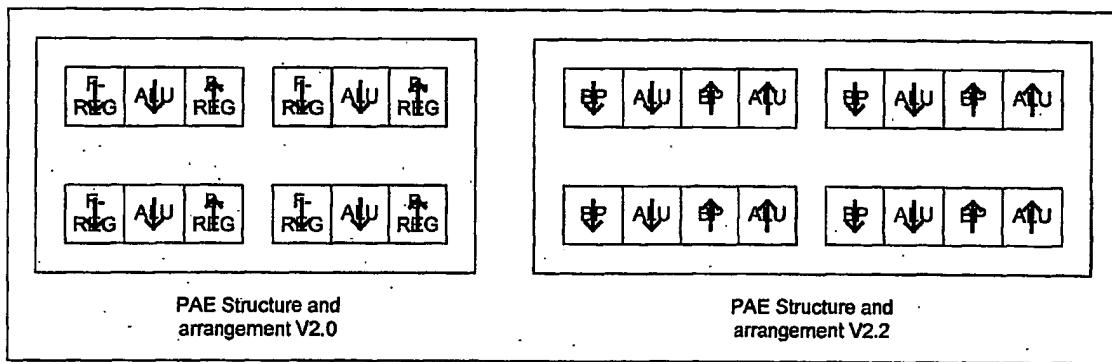
1.4 Bus modifications

Within this chapter optimizations are described which might
reduce the required area and the amount of busses. However,
those modifications comprise several proposals, since they
have to be evaluated based on real algorithms. It is possible
to e.g. compose a questionnaire to collect the necessary input
from the application programmes.

1.4.1 Next neighbour

In XPP-II architecture a direct horizontal data path between
two PAEs block a vertical data bus. This effect increases the
required vertical busses within a XPP and drives cost unneces-
sarily. Therefore in XPP-III a direct feed path between hori-
zontal PAEs is proposed.

In addition horizontal busses of different length are pro-
posed, i.e. next neighbour, crossing 2 PAEs, crossing 4 PAEs.

1.4.2 Removal of registers in busses

In XPP-II registers are implemented in the vertical busses
which can be switched on by configuration for longer paths.
This registers can furthermore be preloaded by configuration
which requires a significant amount of silicon area. It is
proposed to not implement registers in the busses any more,
but to use an enhanced DF or Bypass (PB) part within the PAEs
which is able to reroute a path to the same bus using the DF
or BP internal registers instead:
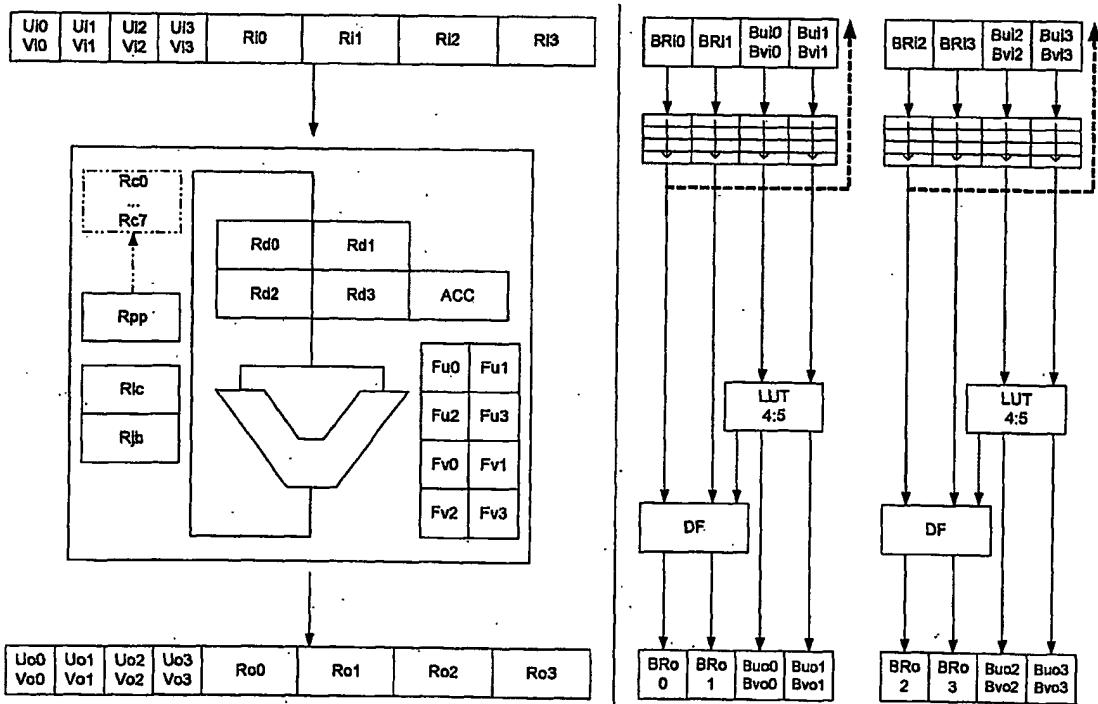
*Here, it might be to decide how many resources are saved for
the busses and how many are needed for the PAEs and /or how
often must registers be inserted, are 1 or max. 2 paths enough
per PAE (limit is two since DF/BP offers max. 2 inputs*

1.4.3 Shifting n:1, 1:n capabilities from busses to PAEs

In XPP-II n:1 and 1:n transitions are supported by the busses which requires a significant amount of resources i.e. for the sample-and-hold stage of the handshake signals.

Depending on the size of n two different capabilities are provided with the new PAE structure:

$n \leq 2$ The required operations are done within the DF path of the PAE

$2 \leq n \leq 4$ The ALU path is required since 4 ports are necessary

$n > 4$ Multiple ALUs have to be combined

This method saves a significant amount of static resources in silicon but requires dedicated PAE resources at runtime.

Here, it might be worthwhile to evaluate how much silicon area is saved per bus how often occurs n=2, $2 \leq n \leq 4$, n > 4 the ratio between saved silicon area and required PAE resource and to decide on the exact bus structure in repsonse to one or all of said criteria.

1.5 FSM in RAM-PAEs

In the XPP-II architecture implementing control structures is very costly, a lot of resources are required and programming is quite difficult.

However memories can be used for a simple FSMs implementation. The following enhancement of the RAM-PAEs offers a cheap and easy to program solution for many of the known control issues, including HDTV.
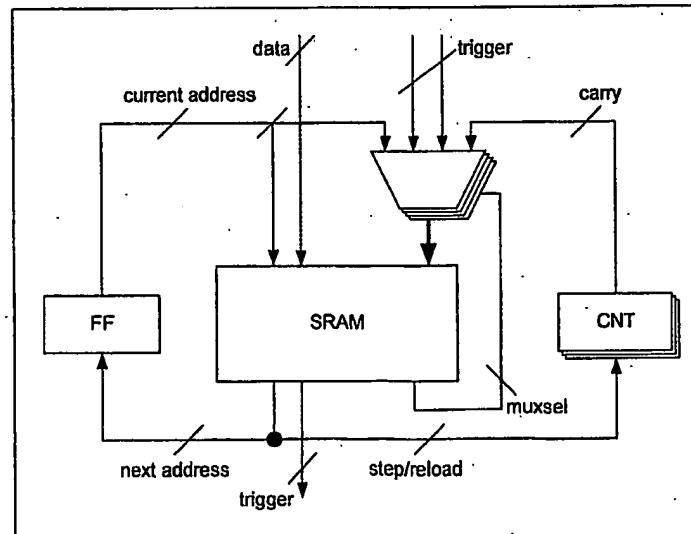
Basically the RAM-PAE is enhanced by an feedback from the data
output to the address input through a register (FF) to supply
subsequent address within each stage. Furthermore additional
address inputs from the PAE array can cause conditional jumps,
data output will generate event signals for the PAE array.
Associated counters which can be reloaded and stepped by the
memory output generate address input for conditional jumps
(i.e. end of line, end of frame of a video picture).
A typical RAM-PAE implementation has about 16-32 data bits but
only 8-12 address bits. To optimize the range of input vectors
it is therefore suggested to insert some multiplexers at the
address inputs to select between multiple vectors, whereas the
multiplexers are controlled by some of the output data bits.

One implementation for an XPP having 24bit wide data busses is
sketched in the next figure. 4 event inputs are used as input,
as well as the lower for bits of input port Ri0. 3 counters
are implemented, 4 events are generated as well as the lower
10 bits of the Ro0 port.

The memory organisation suggested here may be as follows:
     8 address bits
    24 data bits (22 used)
         4 next address
         8 multiplexer selectors
         6 counter control (shared with 4 additional next ad-
dress)
         4 output
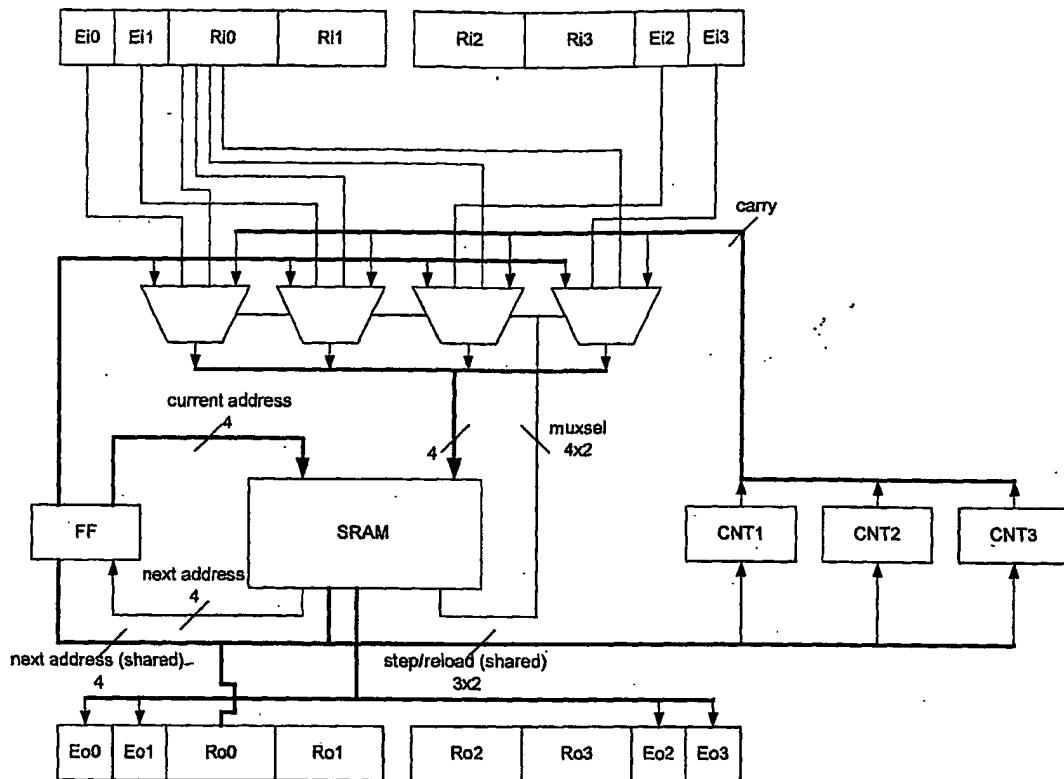
It is to be noted that the typical memory mode of the RAM-PAE is not sketched in the block diagram above.

The width of the counters is according to the bus width of the data busses.

For a 16 bit implementation it is suggested to use the carry signal of the counters as their own reload signal (auto reload), also some of the multiplexers are not driven by the memory but "hard wired" by the configuration.

The proposed memory organisation is as follows:
    8 address bits
    16 data bits (16 used)
        4 next address
        4 multiplexer selectors
        3 counter control (shared with 3 additional next address)
        4 output
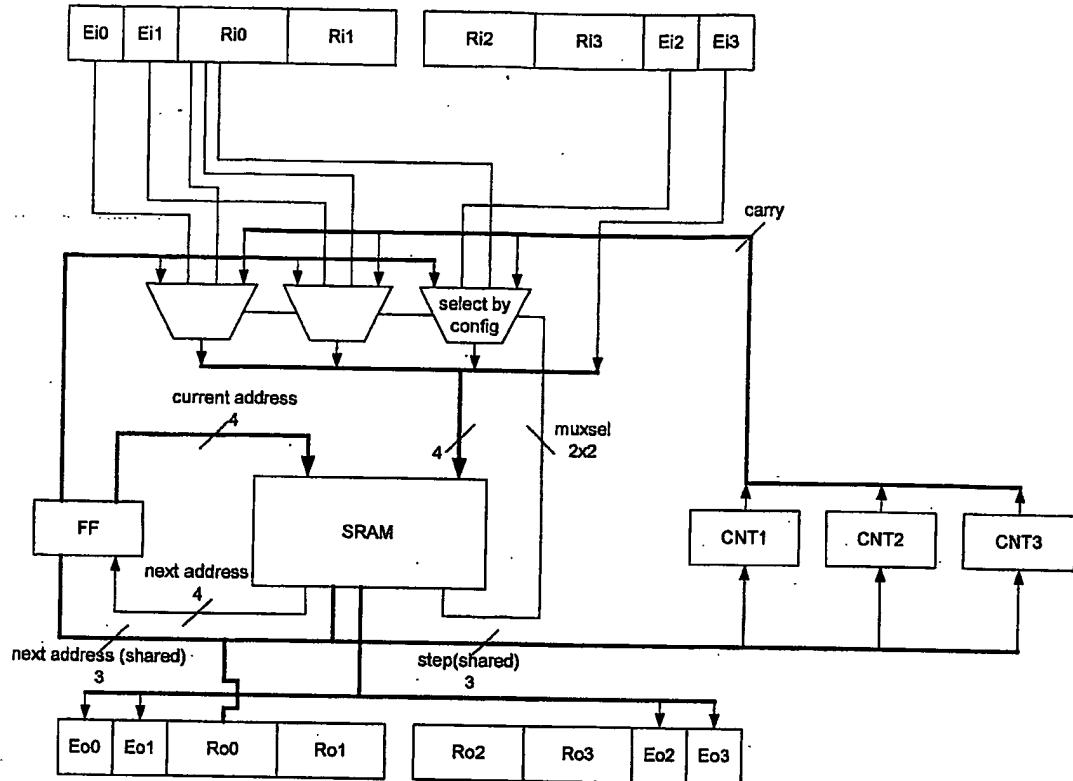
It is to be noted that actually the RAM-PAEs typically will not be scaleable any more since the 16-bit implementation is different from the 24-bit implementation. It is to decide whether the striped down 16-bit implementation is used for 24-bit als

1.6 IOAG interface

1.6.1 Address Generators and bit reversal addressing

Implemented within the IO interfaces are address generators to support e.g. 1 to 3 dimensional addressing directly without any ALU-PAE resources. The address generation is then done by 3 counters, each of them has e.g. configurable base address, length and step width.
The first counter (CNT1) has a step input to be controlled by the array of ALU-PAEs. Its carry is connected to the step input of CNT2, which carry again is connected to the step input of CNT3.
Each counter generates carry if the value is equal to the configured length. Immediately with carry the counter is reset to its configured base address.

One input is dedicated for addresses from the array of ALU-
PAEs which can be added to the values of the counters. If one
or more counters are not used they are configured to be zero.

In addition CNT1 supports generation of bit reversal address-
ing by supplying multiple carry modes.
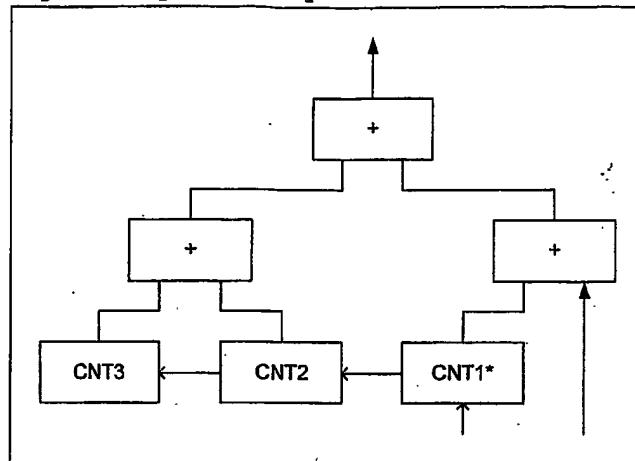
1.6.2 Support for different word width

In general it is necessary to support multiple word width
within the PAE array. 8 and 16 bit wide data words are pre-
ferred for a lot of algorithms i.e. graphics. In addition to
the already described SIMD operation, the IOAG allows the
split and merge of such smaller data words.

Since the new PAE structure allows 4 input and 4 output ports,
the IOAG can support word splitting and merging as follows:

I/O 0	I/O 1	I/O 2	I3
16/24/32-bit data word			address
16-bit data word	16-bit data word		address
8-bit data word	8-bit data word	8-bit data word	address

Input ports are merged within the IOAG for word writes to the
IO.
For output ports the read word is split according to the con-
figured word width.

1.7 Multi-Voltage Power Supply and Frequency Stepping

PAEs and busses are build to perform depending on the work-
load. Therefore the clock frequency is configurable according

to the data bandwidth, in addition clock gating for registers
is supported, busses are decoupled using row of AND gates.
Dynamically clock pulses are gated, whenever no data can be
processed.

Depending on the clock frequency in the PAEs and the required
bandwidth for the busses the voltage is scaled in an advanced
architecture. Within the 4S project such methods are evaluated
and commercially usable technologies are researched.

1.8 XPP / µP coupling

For a closed coupling of a µP and a XPP a cache and register
interface would be the preferable structure for high level
tools like C-compilers. However such a close coupling is ex-
pected not to be doable in a very first step.

Yet, two different kind of couplings may be possible for a
tight coupling:
  a) memory coupling for large data streams: The most conven-
     ient method with the highest performance is a direct
     cache coupling, whereas an AMBA based memory coupling
     will be sufficient for the beginning (to be discussed
     with ATAIR)
  b) register coupling for small data and irregular MAC opera-
     tions: Preferable is a direct coupling into the proces-
     sors registers with an implicit synchronisation in the
     OF-stage of the processor pipeline. However coupling via
     load/store- or in/out-commands as external registers is
     acceptable with the penalty of a higher latency which
     causes some performance limitation.

2 Specification of ALU-PAE

2.1 Overview

In a preferred embodiment, the ALU-PAE comprises 3 paths:
    ALU arithmetic, logic and data flow handling
    BP   bypass

Then, each of the paths contains 2 data busses and 1 event
bus. The busses of the DF path can be rerouted to the ALU path
by configuration.

2.2 ALU path Registers

The ALU path comprises 12 data registers:
    Ri0-3   Input data register 0-3 from bus
    Rv0-3   Virtual output data register 0-3 to bus

RdO-3   Internal general purpose register 0-3

ViO-3   V event input 0-3 from bus
UiO-3   U event input 0-3 from bus

EvO-3   Virtual V event output register 0-3 to bus
EuO-3   Virtual U event output register 0-3 to bus

FuO-3
FvO-3   Internal Flag u and v registers according to the
XPP-II PAE's event   busses

    Acc   Accumulator

Eight instruction registers are implemented, each of them is
24 bit wide according to the opcode format.
    RcO-7   Instruction registers

Three special purpose registers are implemented:
    Rlc   Loop Counter, configured by CM, not accessible
          through ALU-PAE itself. Will be decremented accord-
          ing to JL opcode. Is reloaded after value 0 is
          reached.
    Rjb   Jump-Back register to define the number of used en-
          tries in Rc[0..7]. It is not accessible through
          ALU-PAE itself.
          If Rpp is equal to Rjb, Rpp is immediately reset to
          0. The jump back can be bound to a condition i.e. an
          incoming event. If the condition is missing, the
          jump back will be delayed.
    Rpp   Program pointer

2.3 Data duplication and multiple input reads

Since Function Folding can operate in a purely data stream
mode as well as in a sequential mode (see 1.2) it is useful to
support Ri reads in dataflow mode (single read only) and se-
quential mode (multiple read). The according protocols are de-
scribed below:

Each input register Ri can be configured to work in one of two
different modes:

Dataflow Mode
This is the standard protocol of the XPP-II implementation:
A data packet is taken read from the bus if the register is
empty, an ACK handshake is generated. If the register is not
empty ACK the data is not latched and ACK is not generated.

If the register contains data, it can be read once. Immediately with the read access the register is marked as empty. An empty register cannot be read.

Simplified the protocol is defined as follows:
```
    RDY & empty      → full
                     → ACK
    RDY & full           → notACK

    READ & empty     → stall
    READ & full          → read data
                     → empty
```

Please note: pipeline effects are not taken into account in this description and protocol.

Sequencer Mode
The input interface is according to the bus protocol definition: A data packet is taken read from the bus if the register is empty, an ACK handshake is generated. If the register is not empty ACK the data is not latched and ACK is not generated.
If the register contains data it can be read multiple times during a sequence. A sequence is defined from Rpp = 0 to Rpp = Rjb. During this time no new data can be written into the register. Simultaneously with the reset of Rpp to 0 the register content is cleared an new data is accepted from the bus.

Simplified the protocol is defined as follows:
```
    RDY & empty      → full
                     → ACK
    RDY & full           → notACK

    READ & empty     → stall
    READ & full          → read data

    (Rpp == Rjb)         → empty
```

Please note: pipeline effects are not taken into account in this description and protocol.

2.4 Data register and event handling

Data registers are directly addressed, each data register can be individually selected. Three address opcode form is used, r_t ← r_{s1}, r_{s0}. An virtual output register is selected by adding 'o' behind the register. The result will be stored in r_t and copied

41

to the virtual output register r_v as well according to the rule
op out $(r_v, r_t) \leftarrow r_{s1}, r_{s0}$.
Please note, accessing input and (virtual) output registers
follow the rules defined in chapter 2.3.

source	r_t	Nota-tion
000	0	Rd0
001	1	Rd1
010	2	Rd2
011	3	Rd3
100	0	Ri0
101	1	Ri1
110	2	Ri2
111	3	Ri3

tar-get	r_t	r_v	Nota-tion
000	0	–	Rd0
001	1	–	Rd1
010	2	–	Rd2
011	3	–	Rd3
100	0	0	Ro0
101	1	1	Ro1
110	2	2	Ro2
111	3	3	Ro3

Events are used equal to data registers. All input and inter-
nal events can be addressed directly, output events are used
whenever an 'o' is added behind the event.

etp	ep_t	ep_v	Notation
000	0	–	Fu0, Fv0
001	1	–	Fu1, Fv1
010	2	–	Fu2, Fv2
011	3	–	Fu3, Fv3
100	0	0	Eou0, Eov0
101	1	1	Eou1, Eov1
110	2	2	Eou2, Eov2
111	3	3	Eou3, Eov3

es4/et4	e_t	e_v	Nota-tion
0000	0	–	v0
0001	1	–	v1
0010	2	–	v2
0011	3	–	v3

0100	0	0	vo0
0101	1	1	vo1
0110	2	2	vo2
0111	3	3	vo3
1000	0	–	u0
1001	1	–	u1
1010	2	–	u2
1011	3	–	u3
1100	0	0	uo0
1101	1	1	uo1
1110	2	2	uo2
1111	3	3	uo3

2.4.1 ACCumulator Mode

To achieve low power consumption and for better supporting
DSP-like algorithms an accumulator register is available which
can be addressed by just one set bit for the result register
(ao) and operand register (ai).
For commutative operations always operand register 1 is re-
placed by ai. For non commutative operations as SUBtract oper-
and register 1 selects, whether ai is the first or second op-
erand. Operand register 2 defines the accordingly other oper-
and.

*It is to be noted that it has to be clarified whether a real
ACCumulator mode makes sense or just a MAC-command should be
implemented to handle the multiply accumulate in a single com-
mand consuming two clock cycles with an implicit hidden accu-
mulator access.*

2.4.2 Parameter Stack Mode (PSTACK)

Unused entries in the Opcode Registers Rc can operate as stack
for constants and parameters. At Rpp == 0000 the Rps PStack
registers points to Rjb +1, which means the PStack area starts
immediately behind the last entry in the Opcode register file.

To access the PStack, the FF-PAE must be in the Fast-Parameter
Mode. Each read access to Ri3 is redirected to read from the
PStack, whereas after each read access the pointer incremented
with one. There is no check for an overflow of the PStack
pointer implemented, an overflow is regarded as a program bug.

0000/Base ⟶

code

jb ⟶

Reset PStack

pop

Overflow ->
Programming
Error
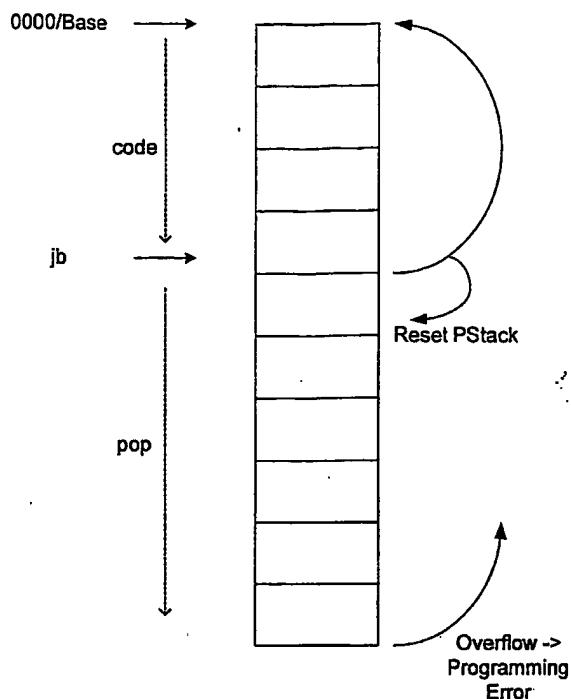
2.4.3 n:1 Transitions

n:1 transitions are not supported within the busses any more.
Alternatively simple writes to multiple output registers Ro
and event outputs Eo are supported. The Virtual Output regis-
ters (Rv) and Virtual Event (Ev) are translated to real Output
registers (Ro) and real Events (Eo), whereas a virtual regis-
ter can be mapped to multiple output registers.

To achieve this a configurable translation table is imple-
mented for both data registers and event registers:

Rv Ev	Ro0 Eo0	Ro1 Eo1	Ro2 Eo2	Ro3 Eo3
0				
1				
2				
3				

Example:
Rv0 mapped to Ro0, Ro1
Rv1 mapped to Ro2
Rv2 mapped to Ro3
Rv3 unused

Rv	Ro0	Ro1	Ro2	Ro3

0	1	1	0	0
1	0	0	1	0
2	0	0	0	1
3	0	0	0	0

2.4.4 Accessing input and output registers (Ri/Rv) and events (Ei/Ev)

Independently from the opcode accessing input or output registers or events is defined as follows:

Reading an input register:

Register status	Operation
empty	wait for data
full	read data and continue operation

Writing to an output register:

Register status	Operation
empty	write data to register
full	wait until register is cleared and can accept new data

2.4.5 Multi-Config Mode

The Multi-Config Mode allows for selecting 1 out of maximum 4 stored configurations. Incoming events on Fui0,1 and Fvi0,1 select one of the 4 configurations. Only one Event shall be active at a clock cycle.
The selection is done by a simple translation, each event points to a specific memory address.
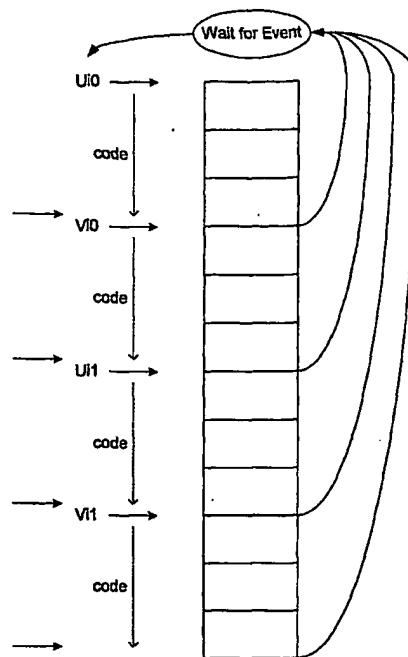
Long configurations may use more than 3 opcode by using the next code segments as well. In this case, the according events can not be used.
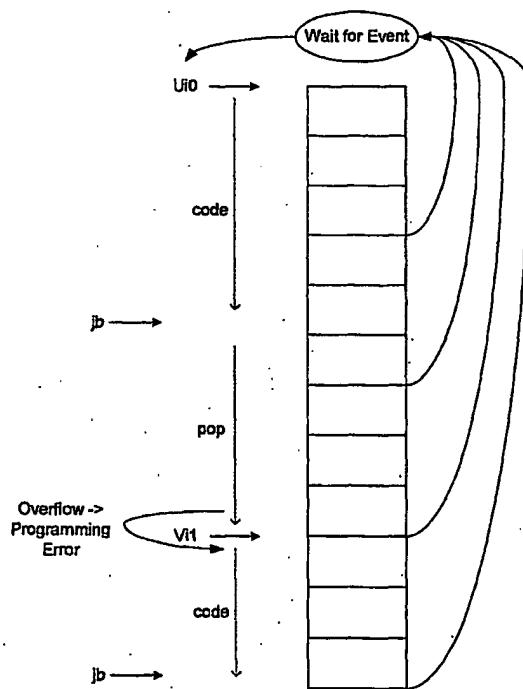
2.5 Opcode format

24 bit wide 3 address opcodes are used in a preferred embodiment:

$$op \ r_t \leftarrow r_a, \ r_b$$

Source registers can be Ri and Rd, target registers are Rv and
Rd. A typical operation targets only Rd registers. If the
source register for r_a is Ri[x] the target register will be
Rd[x].
The translation is shown is the following table:

Target	Source r_a
Rd0	Rd0
Rd1	Rd1
Rd2	Rd2
Rd3	Rd3
Rd0	Ri0
Rd1	Ri1
Rd2	Ri2
Rd3	Ri3

Each operation can target a Virtual Output Register Rv by add-
ing an *out* tag 'o' as a target identifier to the opcode:

$$op \ (r_t, \ ro_t) \leftarrow r_a, \ r_b$$

Data is transferred to the virtual output register and to the
according internal register as well:

Rv	Rd
Rv0	Rd0
Rv1	Rd1
Rv2	Rd2
Rv3	Rd3

2.5.1 Conditional Execution

The SKIPE command supports conditional execution. Either an
event or ALU flag is tested for a specific value. Depending on
the check either the next two addresses are executed (Rpp + 1)
or skipped (Rpp + 3). If an incoming event is checked, the
program execution stops until the event is arrived at the
event port (RDY handshake set).
SKIPE supports conditional execution of any OpCode which is
not larger than two memory entries.
In SEQ-PAEs, which support CALL and RET OpCodes, also stack
based subroutine calls are supported.

2.6 Clock

The PAE can operate at a configurable clock frequency of

```
1x Bus Clock
2x Bus Clock
4x Bus Clock
[8x Bus Clock]
```

2.7 The DF path

The DataFlow path comprises the data registers Bri0..3 and
Bro0..3 as well as the event register Bui/Bvi0..3 and
Buo/Bvo0..3.

The main purpose of the DF path is to establish bus connec-
tions in the vertical direction. In addition the path includes
a 4 stage FIFO for each of the data and event paths.

The DF path supports numerous instructions, whereas the in-
struction is selected by configuration and only one of them
can be performed during a configuration, function folding is
not available.

The following instructions are implemented in the DF path:

1. ADD, SUB
2. NOT, AND, OR, XOR
3. SHL, SHR, DSHL, DSHR, DSHRU
4. EQ, CMP, CMPU
5. MERGE, DEMUX, SWAP
6. SORT, SORTU
7. ELUT

2.9 Parameter Broadcast and Update

Parameters and constants can be updated fast and synchronous
using input register Ri3 and event input Ei7.
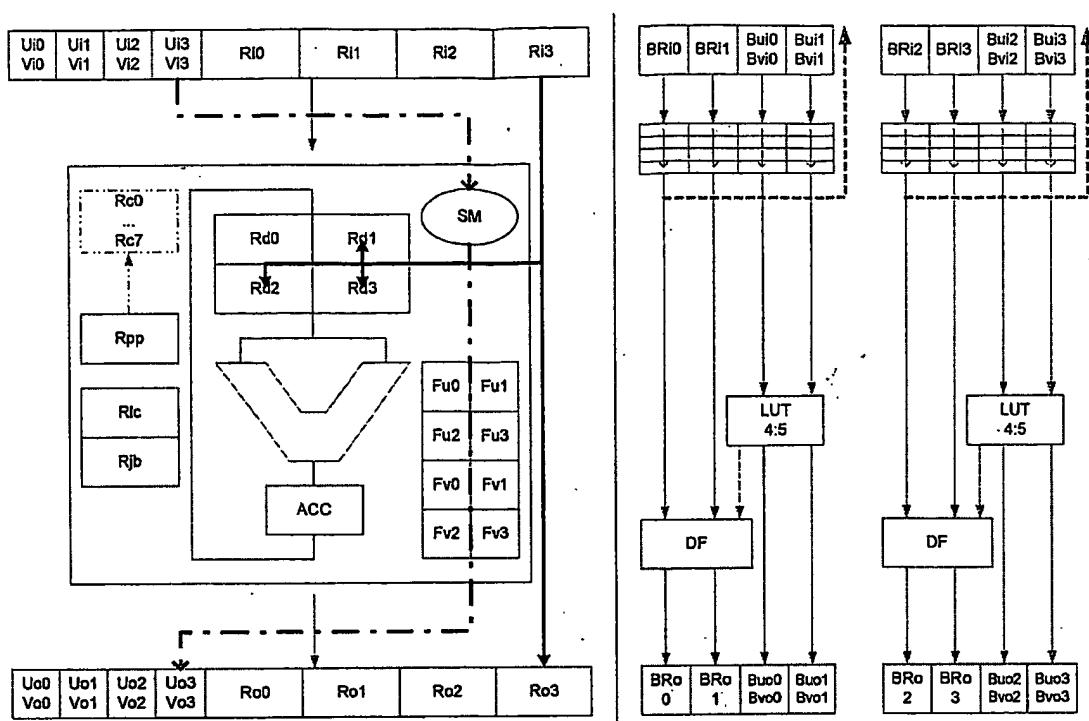
Depending on the update mode, data packets at the input register Ri3 are copied subsequently into Rd3, Rd2 and Rd1 at each access of the according register by the PAE, if the event Ei7 is set. Afterwards all input data at Ri3 is propagated to the output register Ro3, also the Eo7 event output is set, to indicate following PAEs the occurrence of a fast parameter update, which allows to chain PAEs together (i.e. in a multi-TAP FIR filter) and updating all parameters in the chain.

register access	Ei7	UPM1 upmcfg = 0100	UPM2 upmcfg = 1000	UPM3 upmcfg = 1100
–	0	–	–	–
read Rd3	1	Ri3 -> Rd3	Ri3 -> Rd3	Ri3 -> Rd3
read Rd2	1	Ri3 -> Ro3 1 -> Eo7	Ri3 -> Rd2	Ri3 -> Rd2
read Rd1	1	Ri3 -> Ro3 1 -> Eo7	Ri3 -> Ro3 1 -> Eo7	Ri3 -> Rd1
–	1	Ri3 -> Ro3 1 -> Eo7	Ri3 -> Ro3 1 -> Eo7	Ri3 -> Ro3 1 -> Eo7

Also the OpCode UPDATE updates all registers subsequently if Ei7 is set, depending on the Update Parameter Mode (upmcfg = nn10).
Also the register update can be configured to occur whenever Rpp == 0 and Ei7 is set by upmcfg = nn01.

In both cases nn indicates the number of registers to be up-
dated (1-3).

Ei7 must be 0 for at least one clock cycle to indicate the end
of a running parameter update and the start of a new update.

3 Input Output Address Generators (IOAG)

The IOAGs are located in the RAM-PAEs and share, the same reg-
isters to the busses. An IOAG comprises 3 counters with for-
warded carries. The values of the counters and an immediate
address input from the array are added to generate the ad-
dress.
One counter offers reverse carry capabilities.

3.1 Adressing modes

Several addressing modes are supported by the IOAG to support
typical DSP-like addressing:

Mode	Description
Immediate	Address generated by the PAE array
xD counting	Multidimensional addressing using IOAG internal counters xD means 1D, 2D, 3D
xD circular	Multidimensional addressing using IOAG internal counters, after over-flow counters reload with base ad-dress
xD plus immedi-ate	xD plus a value from the PAE array
Stack	decrement after "push" operations increment after "read" operations
Reverse carry	Reverse carry for applications such as FFT

3.1.1 Immediate Addressing

The address is generated in the array and directly fed through
the adder to the address output. All counters are disabled and
set to 0.

3.1.2 xD counting

Counters are enabled depending on the required dimension (x-
dimensions require x counters). For each counter a base ad-
dress and the step width as well as the maximum address are

configured. Each carry is forwarded to the next higher and en-
abled counter; after carry the counter is reloaded with the
start address.
A carry at the highest enabled counter generates an event,
counting stops.

3.1.3 xD circular

The operation is exactly the same as for xD counting, with the
difference that a carry at the highest enabled counter gener-
ates an event, all counters are reloaded to their base address
and continue counting.

3.1.4 Stack

One counter (CNT1) is used to decrement after data writes and
increment after data reads. The base value of the counter can
either be configured (base address) or loaded by the PAE ar-
ray.

3.1.5 Reverse carry

Typically carry is forwarded from LSB to MSB. Forwarding the
carry to the opposite direction (reverse carry) allows gener-
ating address patterns which are very well suited for applica-
tions like FFT and the like. The carry is discarded at MSB.

For using reverse carry a value larger than LSB must be added
to the actual value to count, wherefore the STEP register is
used.

Example:
BASE = 0h
STEP = 1000b

Step	Counter Value
1	b0...00000
2	b0...01000
3	b0...00100
4	b0...01100
5	b0...00010
...	...
16	b0...01111
17	b0...00000

The counter is implemented to allow reverse carry at least for STEP values of -2, -1, +1, +2.

4. ALU/RAM Sequencers ~ SEQ-PAEs

Each ALU-PAE at the left or right edge of the array can be closely coupled to the neighbouring RAM-PAEs as an IP option, thus allowing for configure a sequencer. For compatibility reasons, the data and opcode width of the sequencer is 16bits.

RAM-PAE

ALU-PAE   configurable Sequencer
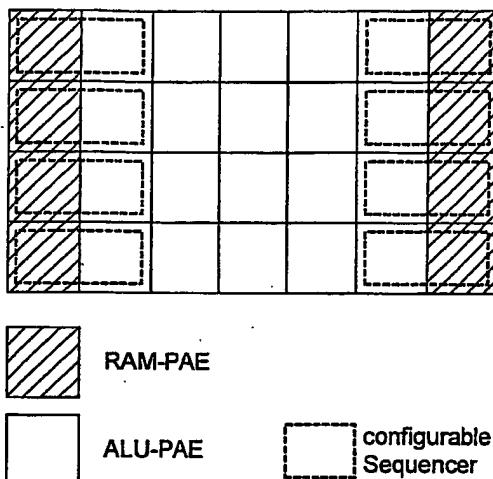
The ALU-PAEs can operate exactly as array internal ALU-PAEs but have several extensions. Operation is Sequencer mode the register file is 8 data registers wide, Fu and Fv flags are used as carry, sign, null, overflow and parity ALU flag word.

Event Registers FF-Mode	Processor Registers SEQ-Mode
Fu0	carry
Fu1	sign
Fu2	null
Fu3	overflow
Fv0	parity

The address width is accordingly 16bit. However since the RAM-PAE size is limited it is segmented into 16 segments. Those segments are used for code, data and stack and must be individually preloaded by the compiler.

4 segment registers point to the specific segments:
      CodeBank   Points to the actual code segment
      DataBank   Points to the actual data segment
      StackBank   Points to the actual stack segment

AuxiliaryBank Points to any segment (but code), allowing
copy   operations between segments

Ui0 Vi0	Ui1 Vi1	Ui2 Vi2	Ui3 Vi3	Ri0	Ri1	Ri2	Ri3

BRi0	BRi1	Bui0 Bvi0	Bui1 Bvi1

BRi2	BRi3	Bui2 Bvi2	Bui3 Bvi3

Rpp

Rspe | Rspl

| Rd0 | Rd1 | Rd2 | Rd3 | ACC |
| Rd4 | Rd5 | Rd6 | Rd7 |

| carry | sign |
| null | over flow |
| parity |

Code Bank
Data Bank e | Data Bank i
Stack Bank e | Stack Bank i
Auxiliary Bank e | Auxiliary Bank i

LUT 4:5

LUT 4:5

DF

DF

Uo0 Vo0	Uo1 Vo1	Uo2 Vo2	Uo3 Vo3	Ro0	Ro1	Ro2	Ro3

BRo0	BRo1	Buo0 Bvo0	Buo1 Bvo1

BRo2	BRo3	Buo2 Bvo2	Buo3 Bvo3

The compiler has to take care that necessary data segments are
preloaded and available. For cost reasons there is no auto-
matic TLB installed.
Also segments have to be physically direct addressed due to
the absence of TLBs. This means that the compiler has to im-
plement range checking functions for according addresses.

Code segments behave accordingly to data segments. The com-
piler has to preload them before execution jumps into them.
Also jumps are physically direct addressed, due to the absence
of TLBs again.

A relocation of any segments is not possible, the mapping is
fixed by the compiler.

The memory layout is shown below. A simple check mechanism is
implemented to validate or invalidate memory segments.

segment
address

reset:
CB = 0000
SB = 1111
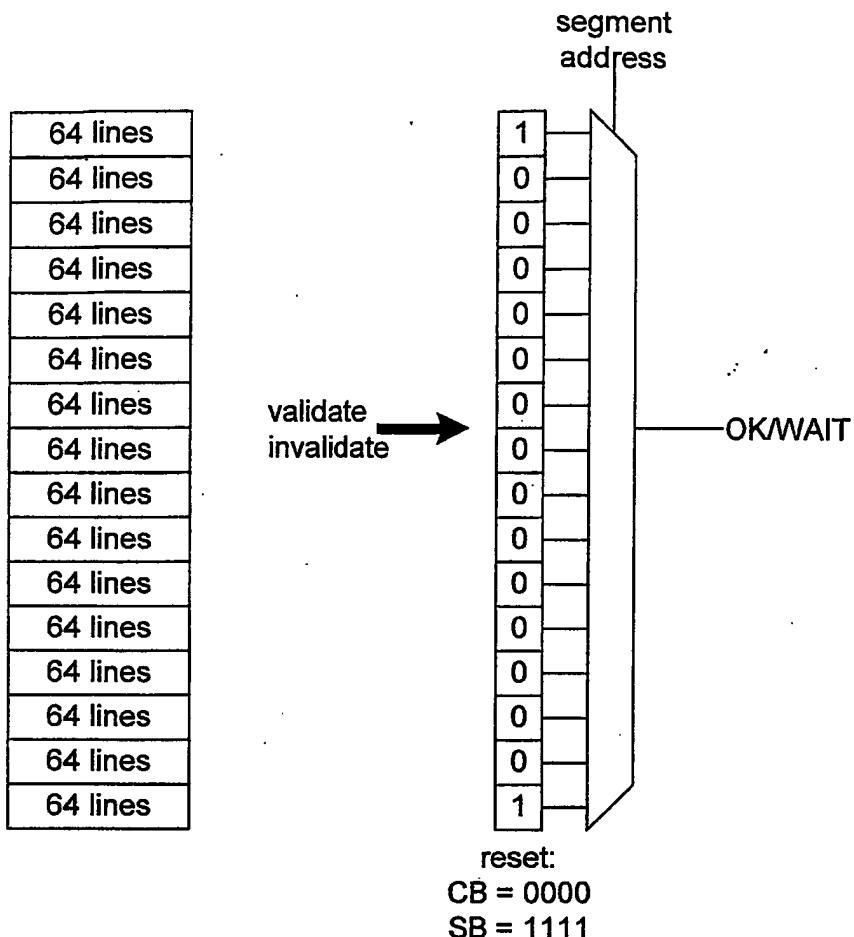
At least the CodeBank (CB) and StackBank (SB) must be set. The
first CodeBank must start at location 0000h. For all other
banks 0000h is an illegal entry. Loading segments to the mem-
ory validates them, accordingly flushing invalidates them.

Memory banks are updates in terms of loaded or flushed in the
background by a DMA engine controlled by the following opcodes
      LOADDSEG   Loads and validates a data/auxiliary/stack
bank
      STOREDSEG Stores and invalidates a data/auxiliary/stack
bank
      LOADCSEG   Loads and validates a code bank
The address generators in the IOAG interfaces can be reused as
DMA engine.

Memory banks can be specifically validated or invalidated as
follows:
      VALIDATESSEG Validates a bank
      INVALIDATESEG Invalidates a bank

The bank pointers are added to the address of any memory ac-
cess. Since the address pointer can be larger than the 6 bits
addressing a 64 line range, segment boarders are not "sharp",

which means, can be crossed without any limitation. However
the programmer or compiler has to take care that no damage oc-
curs while crossing them. If an invalid segment is reached a
flag or trap is generated indicating the fault, eventually
just wait states are inserted if a segment preload is running
already in the background.
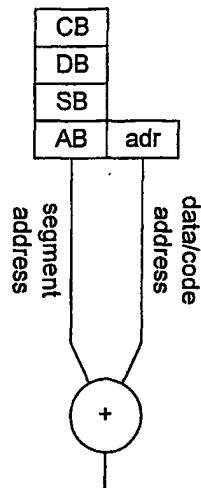
Alternatively a more advanced valid checking scheme can be im-
plemented as shown below:
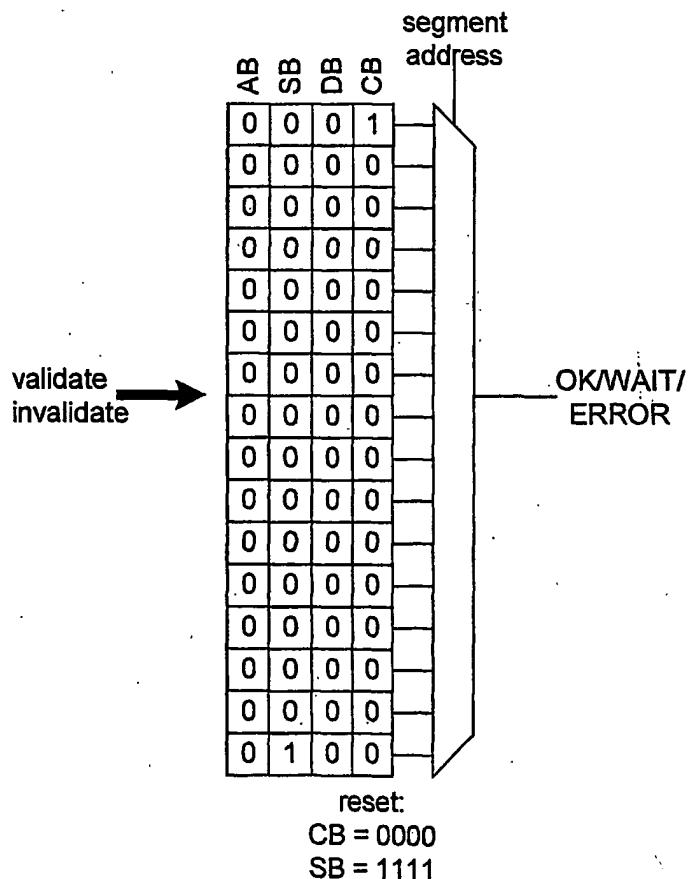
In difference to PAEs which require 24-bit instructions sequencers use 16-bit instructions only. To use the same instruction set and to keep the decoders simple, just the last 8 bits are discarded in sequencer mode.

4.1 IOAGs

IOAGs may comprise a 4-8 stage data output buffer to balance external latency and allow reading the same data address directly after the data has been written, regardless of external bus or memory latencies (up to the number of buffer stages).

In the follwoing, a number of OpCodes and their meanings is suggested:

ADD

ADD

Description:
Add rs1 and rs2.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

et4	
0nnn	F[nnn], F[nnn]
1nnn	Eo[nnn], Eo[nnn]

```
I1, I2 -> O
Rpp++
```

```
        rs:  source register
        rt:  target register
        et4:      target event
```

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
F, Ei

Output Flags:

Mode	
SEQ	carry, sign, null, parity
FF	carry -> Fu / Euo

ADDC

ADD with Carry

Description:
Add rs1 and rs2 with Carry.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Event Input E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

etp	
0nn	Fu[nn], Fv[nn]
1nn	Euo[nn], Evo[nn]

I1, I2 -> O
Rpp++

    rs: source register
    rt: target register
    es4: source event
    etp: target event pair

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
F, Ei

Output Flags:

Mode	
SEQ	carry, sign, null, parity, over- flow
FF	carry -> Fu / Euo, overflow -> Fv / Evo

AND

Logical AND

Description:
Logical AND operation

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

et4	
0nnn	F[nnn], F[nnn]
1nnn	Eo[nnn], Eo[nnn]

I1, I2 -> O
Rpp++

  rs: source register
  rt: target register

<u>Input Registers:</u>
Ri / Rd

<u>Output Registers:</u>
Rd / Ro

<u>Input Flags:</u>
–

<u>Output Flags:</u>

Mode	
SEQ	zero, sign , parity
FF	zero, sign -> F / Eo

BSHL

Barrel SHift Left

<u>Description:</u>
Shift rs1 left by rs2 positions and fill with zeros.

<u>Action:</u>
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O = .

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

I1, I2 -> O
Rpp++

    rs: source register
    rtp: target register pair

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
-

Output Flags:
-

BSHR
Barrel SHift Right

Description:
Shift rs1 right by rs2 positions, sign bit is duplicated.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rtp	

0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

I1, I2 -> O
Rpp++

    rs: source register
    rtp: target register pair

<u>Input Registers:</u>
Ri / Rd

<u>Output Registers:</u>
Rd / Ro

<u>Input Flags:</u>
-

<u>Output Flags:</u>
  -

BSHRU

Barrel SHift Right Unsigned

<u>Description:</u>
Shift rs1 right by rs2 positions and fill with zeros.

<u>Action:</u>
<u>Input I1 =</u>

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

<u>Input I2 =</u>

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

<u>Output O =</u>

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

I1, I2 -> O

Rpp++

     rs: source register
     rtp: target register pair

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
—

Output Flags:
—

CLZ

Count Leading Zeros

Description:
Count the amount of leading zeros if the number is positive, accordingly, count the amount of leading ones if the number is negative.

Action:
Input I1 =

rs1	
Onn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
Onn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

etp	
Onn	Fu[nn], Fv[nn]
1nn	Euo[nn], Evo[nn]

I1 -> O
Rpp++

     rs: source register
     rt: target register

etp: target event pair

Input Registers:
Ri / Rd   .

Output Registers:
Rd / Ro

Input Flags:
─

Output Flags:

Mode	
SEQ	sign, parity, zero
FF	sign, zero -> F / Eo

CLZU
Count Leading Zeros Unsigned

Description:
Count the amount of leading zeros of an unsigned number.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

et4	
0nnn	F[nnn]
1nnm	Eo[nnn]

I1 -> O
Rpp++

    rs: source register
    rt: target register
    et4: target event

Input Registers:

Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
-

Output Flags:

Mode	
SEQ	sign, parity, zero
FF	zero -> F / Eo

CMP

CoMPare

Description:
Compare two values

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Event output Eo =

etp	
0nn	Fu[nn], Fv[nn]
1nn	Euo[nn], Evo[nn]

Rpp++

    rs: source register
    etp: target event pair

Input Registers:
Ri / Rd

Output Registers:
-

Input Flags:

−

Output Flags:

Mode	
SEQ	sign, zero
FF	sign, zero -> F / Eo

CMPU

CoMPare Unsigned

Description:
Compare two unsigned values.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Event output Eo =

etp	
0nn	Fu[nn], Fv[nn]
1nn	Euo[nn], Evo[nn]

Rpp++

rs: source register
etp: target event pair

Input Registers:
Ri / Rd

Output Registers:

−

Input Flags:

−

Output Flags:

Mode	
SEQ	sign, zero
FF	sign, zero -> F / Eo.

DEMUX   **FF**

DEMUltipleX data stream

Description:
Moves input to one of two outputs, depending on flag.

Action:
Input I =

rs	
0nn	Rd[nn]
1nn	Ri[nn]

Output O1 =

rt1	
0nn	Rd[nn]
1nn	Ro[nn]

Output O2 =

rt2	
0nn	Rd[nn]
1nn	Ro[nn]

Event E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

E	
0	O1 = I
1	O2 = I

Rpp++

        rt: target register
        rs: source register
        es4:   source event

Input Registers:

Ri / Rd

Output Registers:
Rd / Ro, Rd / Ro

Input Flags:
Ei / F

Output Flags:
-

DIV   **SEQ**

DIVide

Description:
Divide rs1 by rs2. Result in rtp, reminder in rtp+1.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

I1, I2 -> O
Rpp++

    rs: source register
    rtp: target register pair

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:

Output Flags:
-

DIVU SEQ

DIVide Unsigned

Description:
Divide unsigned rs1 by rs2. Result in rtp, reminder in rtp+1.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

I1, I2 -> O
Rpp++

    rs: source register
    rtp: target register pair

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
-

Output Flags:
-

DSHL
Double SHift Left

Description:
Shift rs1 and rs2 left. LSB is filled with event.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Event Input E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

Output O =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

Event output Eo =

etp	
0nn	Fu[nn], Fv[nn]
1nn	Euo[nn], Evo[nn]

I1, I2 -> O
Rpp++

    rs: source register
    rtp: target register pair
    etp: target event pair

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
F, Ei

Output Flags:

Mode	·
SEQ	MSB(rs1) -> carry, MSB(rs2) -> sign
FF	MSB(rs1) -> Fu / Euo, MSB(rs2) -> Fv / Evo

DSHR
Double SHift Right

Description:
Shift rs1 and rs2 right, sign bit is duplicated.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

Event output Eo =

etp	
0nn	Fu[nn], Fv[nn]
1nn	Euo[nn], Evo[nn]

I1, I2 -> O
Rpp++

    rs: source register
    rtp: target register pair
    etp: target event pair

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
Ei, F

Output Flags:

Mode	
SEQ	LSB(rs1) -> carry, LSB(rs2) -> sign
FF	LSB(rs1) -> Fu / Euo, LSB(rs2) -> Fv / Evo

DSHRU

Double SHift Right Unsigned

Description:
Shift rs1 and rs2 right and fill with event.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Event Input E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

Output O =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

Event output Eo =

etp	
0nn	Fu[nn],

	Fv[nn]
1nn	Euo[nn], Evo[nn]

I1, I2 -> 0
Rpp++

    rs: source register
    rtp: target register pair
    etp: target event pair

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
Ei, F

Output Flags:

Mode	
SEQ	LSB(rs1) -> carry, LSB(rs2) -> sign
FF	LSB(rs1) -> Fu / Euo, LSB(rs2) -> Fv / Evo

EQ

EQual

Description:
Check whether two values are equal.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Event output Eo =

et4	

Onnn	F[nnn]
1nnn	Eo[nnn]

Rpp++

    rs: source register
    et4:   target event

Input Registers:
Ri / Rd

Output Registers:
-

Input Flags:
-

Output Flags:

Mode	
SEQ	zero
FF	zero -> F / Eo

JMP   SEQ

JuMP immediate

Description:
Jump to address defined by immediate constant. CodeBank is
changed according to constant.

Action:
const[0..3] -> CodeBank
const[4..15] -> Rpp

Input Registers:
-

Output Registers:
-

Input Flags:
-

Output Flags:
-

JRI   SEQ

Jump Relative Immediate

Description:
Jump relative to Rpp according to immediate signed constant.
CodeBank is not influenced.

Action:
Rpp + const -> Rpp

Input Registers:
-

Output Registers:
-

Input Flags:
-

Output Flags:
-

JRR   **SEQ**

Jump Relative Register

Description:
Jump relative to Rpp according to signed content of register.
CodeBank is not influenced.

Action:
Rpp + Rd[rbs] -> Rpp

Input Registers:
-

Output Registers:
-

Input Flags:
-

Output Flags:
-

LOAD

LOAD data register with constant

Description:

Loads internal data register or output register with an imme-
diate constant

Action:

rt	
0nn	const -> Rd[nn]
1nn	const -> Ro[nn]

Rpp++

    rt: target register

Input Registers:
-

Output Registers:
Rd /Ro

Input Flags:
-

Output Flags:
-

MERGE   **FF**
_____

MERGE data streams

Description:
Moves one of two inputs to output, depending on flag.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

E	
0	O = I1
1	O = I2

Rpp++

    rt: target register
    rs: source register
    es: source event

Input Registers:
Ri / Rd, Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
Ei / F

Output Flags:
-

MOVE

MOVE internal data register

Description:
Moves content of a register bank register to another internal
register.

Action:
Rd[rbs] -> rd[rbt]
Rpp++

    rbs: register bank source
    rbt: register bank target

Input Registers:
Rd

Output Registers:
Rd

Input Flags:
-

Output Flags:
-

MOVEE

MOVE flag register

Description:
Moves content of a flag register to another flag register.

Action:
F[fs] -> F[ft]
Rpp++

      fs: flag source
      ft: flag target

Input Registers:
-

Output Registers:
-

Input Flags:
F

Output Flags:
F

MUL

MULtiply

Description:
Multiply rs1 and rs2.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	

0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rtp	.
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

I1, I2 -> O
Rpp++

    rs: source register
    rtp: target register pair

<u>Input Registers:</u>
Ri / Rd

<u>Output Registers:</u>
Rd / Ro

<u>Input Flags:</u>
—

<u>Output Flags:</u>
—

<u>MULU</u>

MULtiply Unsigned

<u>Description:</u>
Multiply unsigned rs1 and rs2.

<u>Action:</u>
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rtp	
0n	Rd[(n*2)],

	Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

```
I1, I2 -> O
Rpp++
```

```
    rs:   source register
    rtp:      target register pair
```

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
-

Output Flags:
-

NOP
No Operation

Description:
No Operation, Rpp is incremented

Action:
Rpp++

Input Registers:
-

Output Registers:
-

Input Flags:
-

Output Flags:
-

NOT
Logical inverse

Description:

Inverts register logically

Action:

Input I =

rs	.
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

I -> O
Rpp++

  rs: source register
  rt: target register

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
-

Output Flags:

Mode	.
SEQ	zero
FF	F / Eo

OR

Logical OR

Description:
Logical OR operation

Action:

Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]

1nn	Ri[nn]

Output O =

rt	·
0nn	Rd[nn]
1nn	Ro[nn]

I1, I2 -> O
Rpp++

      rs: source register
      rt: target register

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
—

Output Flags:

Mode	
SEQ	zero, sign , parity
FF	zero -> F / Eo

READptr

READ data input register

Description:
Read specified data input register and write to internal reg-
ister bank or output register. READ waits until data is avail-
able at the input register.

Action:

rt	
0nn	Ri[ri] -> Rd[nn]
1nn	Ri[ri] -> Ro[nn]

Rpp++

      rt: target register

ri: input register

Input Registers:
Ri

Output Registers:
Rd / Ro

Input Flags:
-

Output Flags:
-

READE
READ event input register

Description:
Read specified event input register and write to internal flag
bank or event output register. READE waits until event is
available at the input register.

Action:

et4	
0nnn	Ei[ei] -> F[nnn]
1nnn	Ei[ei] -> Eo[nnn]

Rpp++

    et4:   target event
    ei: input event

Input Registers:
-

Output Registers:
-

Input Flags:
Ei

Output Flags:
F / Eo

SAT

SATurate

Description:
Saturates register depending on carry (Fu0) flag and satura-
tion mode..

Action:
Input I =

rs	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event E=

Mode	es4	
SEQ	don't care	carry
FF	0nnn	F[nnn]
FF	1nnn	Ei[nnn]

E	as	
0	don't care	I -> O
1	0	0h -> O
1	1	ffffh -> O

Rpp++

        rs: source register
        rt: target register
        as: add/substract mode
        es4: event source

Input Registers:
Rd

Output Registers:
Rd / Ro

es4 Input Flags:
SEQ-Mode :   carry
FF-Mode: Ei/F

Output Flags:
-

SETF

SET Flag with constant

Description:
Loads flag register or output event with an immediate constant

Action:

et4	
0nnn	const -> F[nnn]
1nnn	const -> Eo[nnn]

Rpp++

    et4:   event target

Input Registers:
-

Output Registers:
-

Input Flags:
-

Output Flags:
F /Eo

SHL

SHift Left

Description:
Shift rs1 left. LSB is filled with event.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Event Input E=

es4	

| 0nnn | F[nnn] |
| 1nnn | Ei[nnn] |

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

et4	
0nnn	F[nnn]
1nnn	Eo[nnn]

I1 -> O
Rpp++

     rs: source register
     rt: target register pair
     et4:   target event pair
     es4: source event register

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
F, Ei

Output Flags:

Mode	
SEQ	MSB(rs1) -> carry
FF	MSB(rs1) -> Fu / Euo

SHR
SHift Right

Description:
Shift rs1 right. MSB is filled with event.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Event Input E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

et4	
0nnn	F[nnn]
1nnn	Eo[nnn]

```
I1 -> O
Rpp++
```

    rs: source register
    rt: target register pair
    et4:   target event pair
    es4: source event register

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
F, Ei

Output Flags:

Mode	
SEQ	LSB(rs1) -> carry
FF	LSB(rs1) -> Fu / Euo

SKIPE

SKIP next two commands depending on Event

Description:
Next two commands are skipped based on event or flag. If an
event is selected as source the execution stops until the
event is available.

Action:

val	value
0	0
1	1

Event E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

Skip next two addresses if event or flag is equal to val:

event/flag	
not equal val	Rpp++
equal val	Rpp + 3 -> Rpp

    val: value
    es4: event source

Input Registers:
-

Output Registers:
-

Input Flags:
Ei / F

Output Flags:
-

SORT   **FF**

SORT data stream

Description:
Sort two inputs, depending on value.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O1 =

rt1	
0nn	Rd[nn]
1nn	Ro[nn]

Output O2 =

rt2	
0nn	Rd[nn]
1nn	Ro[nn]

Event E1=

et41	
0nnn	F[nnn]
1nnn	Eo[nnn]

Event E2=

et42	
0nnn	F[nnn]
1nnn	Eo[nnn]

O1 = smaller value of I1 and I2
O2 = larger value of I1 and I2

E1 = 1 if I1 < I2 else 0
E2 = 1 if I1 <= I2 else 0

Rpp++

        rt: target register
        rs: source register
        et4:   target event

Input Registers:
Ri / Rd, Ri / Rd

Output Registers:
Rd / Ro, Rd / Ro

Input Flags:
-

Output Flags:
Ei / F

SORTU   **FF**

SORT data stream Unsigned

Description:
Sort two unsigned inputs, depending on value.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O1 =

rt1	
0nn	Rd[nn]
1nn	Ro[nn]

Output O2 =

rt2	
0nn	Rd[nn]
1nn	Ro[nn]

Event E1=

et41	
0nnn	F[nnn]
1nnn	Eo[nnn]

Event E2=

et42	
0nnn	F[nnn]
1nnn	Eo[nnn]

O1 = smaller value of I1 and I2
O2 = larger value of I1 and I2

E1 = 1 if I1 < I2 else 0
E2 = 1 if I1 <= I2 else 0

Rpp++

        rt: target register
        rs: source register
        et4:   target event

Input Registers:
Ri / Rd, Ri / Rd

Output Registers:
Rd / Ro, Rd / Ro

Input Flags:
‐

Output Flags:
Ei / F

SUB

SUBtract

Description:
Subtract rs2 from rs1.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

et4	
0nnn	F[nnn], F[nnn]
1nnn	Eo[nnn], Eo[nnn]

I1, I2 -> O
Rpp++

 rs: source register
 rt: target register
 et4: target event

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
F, Ei

Output Flags:

Mode	
SEQ	carry, sign, null, parity
FF	carry -> Fu / Euo

ADDC
ADD with Carry

Description:
Subtract rs2 from rs1 with Carry.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Event Input E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

etp	
0nn	Fu[nn], Fv[nn]
1nn	Euo[nn],

	Evo[nn]

```
I1, I2 -> O
Rpp++
```

```
        rs:  source register
        rt:  target register
        es4: source event
        etp:     target event pair
```

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
F, Ei

Output Flags:

Mode	
SEQ	carry, sign, null, parity, over-flow
FF	carry -> Fu / Euo, overflow -> Fv / Evo

SWAP   FF

SWAP data stream

Description:
Swap two inputs, depending on flag.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O1 =

rt1	
0nn	Rd[nn]
1nn	Ro[nn]

Output O2 =

rt2	
0nn	Rd[nn]
1nn	Ro[nn]

Event E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

E	
0	O1 = I1, O2 = I2
1	O1 = I2, O2 = I1

Rpp++

     rt: target register
     rs: source register
     es4:   source event

Input Registers:
Ri / Rd, Ri / Rd

Output Registers:
Rd / Ro, Rd / Ro

Input Flags:
Ei / F

Output Flags:
—

UPDATE   FF

UPDATE parameters

Description:
Updates registers Rd3, Rd2, Rd1 with value from Ri3, if Ei7 is
set. Moves subsequent data packet on Ri3 to Ro3 and sets Eo7.

Action:

Mode				
1	Ri3 -> Rd3	set Eo7 Ri3 -> Ro3		
2	Ri3 -> Rd3	Ri2 -> Rd2	set Eo7 Ri3 -> Ro3	
3	Ri3 -> Rd3	Ri2 -> Rd2	Ri1 -> Rd1	set Eo7

				Ri3 -> Ro3

Rpp++

    mode: ·   update mode

<u>Input Registers:</u>
Ri3

<u>Output Registers:</u>
Rd3, Rd2, Rd1

<u>Input Flags:</u>
Ei7

<u>Output Flags:</u>
Eo7

WAITE
WAIT for incoming Event

<u>Description:</u>
Stop execution and wait for incoming event of defined value.
Acknowledge incoming events.

<u>Action:</u>

valx	value
00	0
01	1
1x	don't care

Event E= .

es3	
nnn	Ei[nnn ·]

Wait for incoming event of defined value. Acknowledge all in-
coming events.

    valx: value
    es3: event source

Rpp++

<u>Input Registers:</u>
-

<u>Output Registers:</u>

Input Flags:
Ei

Output Flags:
-

WRITE

WRITE output register

Description:
Write data from input register or internal register bank to
output register. Wait for incoming ACK either before or after
writing.

Action:
<sync0>

rs	
0nn	Ri[nn] -> Ro[ro]
1nn	Rd[nn] -> Ro[ro]

<sync1>
Rpp++

    ro: output register
    rs: register source

Synchronisation is handled according to sy:

sy = 0 <sync0>	Wait only if previously sent event has not been granted by ACK yet
sy = 1 <sync1>	Wait until actual event is granted by ACK

Input Registers:
Ri / Rd

Output Registers:
Ro

Input Flags:
-

Output Flags:
-

WRITEE

WRITE Event output register

Description:
Write event from input register or flag to event output regis-
ter. Wait for incoming ACK either before or after writing.

Action:
<sync0>

es4	
0nnn	Ei[nnn] -> Eo[eo]
1nnn	F[nnn] -> Eo[eo]

<sync1>
Rpp++

      eo: output event
      es4: event source

Synchronisation is handled according to sy:

sy = 0 <sync0>	Wait only if previously sent event has not been granted by ACK yet
sy = 1 <sync1>	Wait until actual event is granted by ACK

Input Registers:
-

Output Registers:
-

Input Flags:
Ei / F

Output Flags:
Eo

XOR

Logical XOR

Description:
Logical XOR operation

Action:

Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

I1, I2 -> O
Rpp++

        rs: source register
        rt: target register

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
-

Output Flags:

Mode	
SEQ	zero, sign , parity
FF	zero -> F / Eo

Appendix B

In the following, an exaple for the use of function folding is given:

Function Folding and Fast Parameter Update Example FIR

Ri0 = x
Ri1 = y

3-folded FIR using acc
Fast parameter update for registers Rd1, Rd2, Rd3

example 1: UPM3, updates parameters with each access to
Rd3,2,1 (if Ei7 is set)
upmcfg = 1100
stage 1
```
      mul acc, Ri0, Rd3;
      add Rd0, acc, Ri1;
```
stage 2
```
      mul acc, Ri0, Rd2;
      add Rd0, acc, Rd0;
```
stage 3
```
      mul acc, Ri0, Rd1;
      add Ro1, acc, Rd3;
      write Ro0, Ri0;
```

Alternative using MAC opcode, parameter pop and looping
```
          read Rd0, Ri1;
lh,lt[3]:     mac Rd0, Ri0, pop;
              write Ro1, Rd0;
          write Ro0, Ri0;
```

example 2: UPM3, uses command UPDATE for parameter update
upmcfg = 1110
stage 1
```
      mul acc, Ri0, Rd3;
      add Rd0, acc, Ri1;
```
stage 2
```
      mul acc, Ri0, Rd2;
      add Rd0, acc, Rd0;
```
stage 3
```
      mul acc, Ri0, Rd1;
      add Ro1, acc, Rd3;
      write Ro0, Ri0;
      update 3
```

example 3: UPM3, updates parameters at Rpp == 0
upmcfg = 1101
stage 1
```
      mul acc, Ri0, Rd3;
      add Rd0, acc, Ri1;
```
stage 2
```
      mul acc, Ri0, Rd2;
      add Rd0, acc, Rd0;
```
stage 3
```
      mul acc, Ri0, Rd1;
      add Ro1, acc, Rd3;
      write Ro0, Ri0;
```

In the above, an improved data processor array has been de-
scribed. Although only in some instances, it has been pointed
out that reference to a certain number of registers, bit width
etc. is for explanation only, it is to be understood that this
also holds where such reference is not found.

If the array is to be very large or in case a real time

process is run where two different fragments of an array un-

known at compile time have to communicate with each other so

as to enable data processing, it is advantageous to improve

the performance by ensuring that a communication path can be

set up. Several suggestions have been made already, e.g. Lee-

Routing and/or the method described in PACT 7. It is to be un-

derstood that the following part of an improved array design

might result in an improved circuitry for certain applications

but that it is not deemed absolutely and inevitably necessary

to implement it with e.g. a function fold PAE. Rather, the

other suggestions for improvement will result in significant

improvements on their own as will be understood by the average

skilled person.

ROUTING IMPROVEMENT

The suggested improvement described hereinafter concerns the static routing
network for reconfigurable array architectures. Hereby this static network
is enhanced by implementing additional logic to adaptive runtime routing.

Figure 1 depicts a cut-out of a reconfigurable array with a set of functional units (FU). Each functional unit encloses one routing unit (RU) and additional functional modules (FMs). The enclosed functional modules are used to manipulate data and characterize the type of the FU. The RU contains an interconnect matrix which is able to route each input port to any desirable output ports. All FUs are connected through point-to-point links whereas each is composed of two half-duplex links and able to transport the data in both directions at the same time.

The routing technique described in this document is instruction based which means that each routing process must be started by an instruction. If the user wants to establish a routing between two cells, he has to bring a specific instruction into the source cell. The hardware within the array calculates based on the instruction fields values the desired routing direction and establishes the logic stream. The routing process happens stepwise from one functional unit to another whereby each cell decides which direction should be taken next. On the way to an established route we defined three valuable states of the routing resources. The first state is the physical route or link. This means that the resources of this route are not used and available to routing processes. The second state is named temporal route or link. This state describes the temporarily not available link, which means that this link is in use for routing purposes but the mentioned routing is not confirmed yet. The problem here is that this route can be confirmed in the future or released if the successor cells are able to realise the desired routing. The last state is the logical route or link. This state represents an established route on the array which is able to transport calculation data.

This routing technique uses coordinates on the array to calculation routings. Each FU possesses unique coordinate's und on the basis of this information it is able to determine the routing direction to each desired cell within the array. This concept is the basis for the adaptive runtime routing described in this document. The needed control logic for adaptive routing is implemented within the routing unit, especially within the routing controller which controls the interconnect matrix at runtime. Therefore the routing controller is able to analyze the incoming data of all input ports of the concerned FU and come to a decision what to do next.

Routing Establishment

For the purpose of incoming data analyzing and data buffering each input port owns so called in-registers (InReg). Additional to those standard registers there are InReg-controllers implemented (InRegCtrl). Those finite state machines (FSMs) have the job to store the actual state of the input links and in dependency of the actual state to trigger routing requests or release not required routings. To fulfil its job each InRegCtrl is connected to an in-controller (InCtrl) which is implemented exactly once per FU. Important requirement for requesting of new routings is that the mentioned input resource (InReg, InRegCtrl) are not used and so in the state of physical link.

InCtrl gets requests of all InRegCtrls all over the time and forwards one request after another to the routing controller (RoutCtrl). The selection which InRegCtrl should be served first is dependant on the routing priority of the input link and/or which input link was served last. Based on the coordinate information of the target cell and the coordinates of the actual FU the RoutCtrl calculates the forward direction for the requested input link. Thereby the RoutCtrl takes into account additional parameters like optimum bit (will be described later), the network utilisation towards the desired direction, etc.

If the direction calculation within the RoutCtrl was successful the RoutCtrl forwards the request with additional information about the output port to the interconnect matrix, which connects the input port with calculated output port. If this is done the RoutCtrl signals the successful

routing operation to InCtrl. Because the actual reached routing state is not final it is necessary to store the actual state. This happens within the queue-request-registerfile (QueueRRF). Therefore the InCtrl is directly connected to the QueueRRF and is able to store the desired information. At this point the related input and output links reach the temporal link state and are temporarily not available for other routing processes.

Due the fact that the QueueRRF is able to store more than one routing entry, the InCtrl is able to hold multiple routing processes at the same time. But for the purpose of high hardware area consumption the direction calculation is realized once within the RoutCtrl.

The established temporal routing stays stored within the QueueRRF till the point the successor cell acknowledges the routing. In this case the InCtrl clear the according entry in the QueueRRF and signals the·successful routing to the InCtrl. The InRegCtrl changes into the state logical route and signal the predecessor cell the successfully finished routing process.

The other case can happen if the successor cell is not able to establish the desired route. In this case the InCtrl forwards a new request to the RoutCtrl based on the QueueRRF-entry. This request leads to new routing suggestion which will be stored within the QueueRRF.

If all available and expedient directions are checked and routing trials failed the InCtrl signals to InRegCtrl the failed routing. The InCtrl signals the same routing miss to the predecessor cell and finishes the routing process in the current cell.

Within the routing process there are two exceptions how the routing unit establishes a desired routing. Those exceptions affect the source and the target cell. The exception in both cases is that as well the source cell as the target cell do not need to route the started/ending routing through the interconnect matrix. To connect the FMs to the output links of cells simple multiplexers are used. Those multiplexers are implemented after the interconnect matrix and have to be switched explicitly. This happens after the routing process is finished. The exception lies in the finishing state. Here the InRegCtrl doesn't have to acknowledge the successful routing the predecessor it just has to consume the actual routing instruction in the InReg instead. This happens after the InCtrl signals the successful routing. Additionally the InReg switches the output multiplexer associated to the output port of the FM and finishes the routing establishment. The information needed the switch the right output multiplexer gets the InCtrl from the RoutCtrl.

Otherwise if the routing fails the InCtrl asserts cell specific interrupt line and signals-the failure to the system.

The second exception concerns the target routing cell. Here it is important to connect the new route with the input ports of the local FM. Therefore simple multiplexers are used which are implemented before the interconnect matrix. If an ongoing routing process reaches the target cell the InCtrl identifies the target achievement and switches the associated input multiplexer to forward the incoming data to the input port of the FM. This is the point where the successful route establishment signal is generated by the InRegCtrl after InCtrl signals the success. Here the InRegCtrl has the last job to finish the routing process by deleting the routing instruction and going to logical state.

Releasing Established Routing

For releasing of the logically established routings we introduced special instructions, so called end packets. The only purpose of those instructions is the route-dissolving by inject the necessary end packet into the logic established routing. There are two ways how the routings can be released. The first possibility is the global releasing. This means that all routes which are following the route where the end packet is injected will be released. This function is useful to delete whole configurations with one single instruction. For this purpose it is important that the FMs are able to forward the end packet unaltered through the internal datapaths.

The second way for route releasing is the local route releasing. Here it is possible to release single established routes between output and input ports of FMs. The end packets are not propagated through the FMs. In this case the end packet will be consumed by the last InRegCtrl.

The internal RU communication is similar to the routing process. If the InRegCtrl determines incoming end packet and the InRegCtrl is in the logic route state, the InRegCtrl forwards the route release request to the InCtrl. The InCtrl clears the entries either within the interconnect matrix or within the input multiplexers registers or within the output multiplexer registers. Meanwhile the InRegCtrl consumes (in case of the local end packet and last cell in the chain) the instruction and goes to the idle state. If the end packet was a global instruction the InRegCtrl forwards alway the end packet to the successor.

Additional Features

For the purpose of priority control, we introduced a priority system to influence the order in which the RU serves the incoming routing requests. Therefore the instructions contain priority fields which describe the priority level. Higher values in this field result in higher priority und will be preferred by the RU during the runtime routing. The priority field has direct influence on the selection of the incoming routing requests from the InRegCtrls to InCtrl.

Some inner configuration communication streams require strictly defined latency to reach the desired performance. Therefore it is very important to keep the maximum register chain length. To decrease the latency of the routed streams its is necessary to ensure that the array chose always the best routing between source and target, but this requirement may lead to not routable streams if this feature will be always required. To ease this problem we introduced a special bit within the routing instruction, so called optimum bit (OptBit). This bit has to be activated if the optimum routing is definitely required. In this case the array tries to reach this requirement und delivers an interrupt if fails.

The alternative to reach the required latency is the speed path counter. This counter gives the possibility to bypass a specific number of registers before buffering again. Therefore we defined a reference value and the counter value. Both numbers are stored within the instruction field. Each passed cell respective the RU compares the counter value and the reference value. If both values are equal then the actual cell buffers the stream and resets the counter. If the counter is smaller than the reference value the current buffer will be bypassed and the counter incremented by one. In this way it is possible to bypass a number of buffers which equals exactly to reference value.

Multi-grained Communication Links

In addition to the coarse-grained point-to-point links we introduced more flexible multi-grained point-to-point links. Hereby one single point-to-point link connects two neighbor cells respective the RUs within those cells. One coarse-grained link consists of a set of wires, e.g. 32 wires for one 32 link, and additionally protocol signals. The whole vector is handled by a single set of control signals which makes this communication resource not usable for multi-grained communication.

To reach this requirement we divided the whole 32 bit vector into single strips, e.g. with groups of 8 times 1 bit strips and 3 times 8 bit strips. Each strip obtained separate control signals and is able to operate independently from other strips.

The idea behind this division is to combine those strips to logical multi-grained sub-links. If you have one multi-grained link you can use the whole vector as one interrelated 32 bit vector or split the whole vector into sub-channels. In this configuration each strip can be one single sub-channel or a group of strips can be gathered to a single sub-channel of desired bit-width. You just have - in respect of hardware costs - to consider that one sub-channel has to fit into one multi-grained link.

Multi-grained Routing

In order to route multi-grained channels it's necessary to use the coarse
grained links to support the routing process. The idea is to route two
links in parallel, one coarse-grained link to support multi-grained routing
and one multi-grained link, which will contain the final multi-grained
stream. Therefore we defined a two packet routing instruction with needed
data fields. The first instruction packet contains - compared to coarse-
grained routing instruction - additional bit mask to specify used multi-
grained sub-links and multi-grained link ID to identify the associated
multi-grained link. The other features like described above - optimum bit,
speed path, priority routing - are support in this routing mode as well.
The routing process within the RU is performed similar to the coarse-
grained routing.
The first packet which arrives in a cell is analyzed by the InRegCtrl and a
request is generated and forwarded to the InCtrl. InCtrl forwards the re-
quest to the RoutCtrl and wait for the acknowledgement. If RoutCtrl finds
one possible routing direction, the InCtrl gets the successful acknowledge-
ment and the temporal routing will be established by the RoutCtrl. Next,
the actual job will be stored within the QueueRRP and the InCtrl waits for
the acknowledgement from the successor cell. If RoutCtrl is not able to
find a possible routing, the InCtrl gets negative acknowledgement and which
will be forwarded to the associated InRegCtrl, which generates the route
unable signal to the predecessor cell and quits the routing process within
this cell.
If the successor cell signals successful routing, the InRegCtrl clears the
related entry in the QueueRRP and finishes the routing. If the successor
cell is not able to establish a rout to the destination cell, it generates
negative acknowledgement signal. Hereupon, the InCtlr starts new request to
the RoutCtrl and handle the responses as described above.
The difference between the coarse-grained routing and multi-grained routing
lies in the handling of the multi-grained interconnect matrix. Each strip
of a multi-grained link is handled separately. The RoutCtrl forwards the
switch request to the strip matcher. Strip matcher has the job to analyze
the input strips and to match them to the output link according to already
used strips. What strip matcher is doing is to map the problem of strip
matching into the time domain and switches the needed switchboxes for each
strip separately one after another.

Routing packet for coarse-grained streams:

| 3 3 2 2 | | 1 1 1 1 1 0 0 | | |
| 1 0 9 8 | | 4 3 2 1 0 9 8 | | |
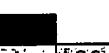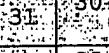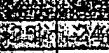
	Value	Comments
	1	instruction-packet
31 30	00	ID: Routing-packet for coarse-grained streams
29 28	XX	Priority-level: higher value results in higher priority
	XX	Speed path: Reference value
	XX	Speed path: Counter
23	X	Optimum bit (OptBit): 1 enabled; 0 disabled
...	XXXX	FM output address within the source cell
...	XXXX	FM input address within the destination cell
14	X	Use fine-grained links: 1 = yes, 0 = no
13 ... 8		Reserved
...	X...X	Destination cell coordinates: x-coordinate
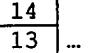
X..X   Destination cell coordinates: y-coordinate

Routing Instruction for multi-grained streams (first packet):

3	3	2	2				2	2	2	2											0	0	0	0
1	0	9	8				5	4	3	2											3	2	1	0

	Value	Comments
	1	Instructions-packet
31 30	01	ID: Routing-instruction multi-grained streams (first packet)
29 28	XX	Priority-level: higher value results in higher priority
	XX	Speed path: Reference value
25 24	XX	Speed path: Counter
23	X	Optimum bit (OptBit): 1 enabled; 0 disabled
22	–	Reserved
	XXX	ID of the input stream of the multi-grained link
	XXX	8 bit strips mask: 1 = selected; 0 = not selected
	X...X	1 bit strips mask: 1 = selected; 0 = not selected
	XXXX	Destination cell coordinates: x-coordinate
	XXXX	Destination cell coordinates: y-coordinate

Second packet of the routing instruction for multi-grained streams:

3	3	2	2																			
1	0	9	8																			

	Value	Comments
	1	Instructions-packet
31 ... 30	10	ID: Routing-instruction multi-grained streams (first packet)
29 28	X	Reserved
	XXX	Destination cell 8 bit strips mask: 1 = selected; 0 = not selected
	X..X	Destination cell 1 bit strips mask: 1 = selected; 0 = not selected
	XXX	Multi-grained FM input port address of the destination cell
13 ... 11	XXX	Source cell 8 bit strips mask: 1 = selected; 0 = not selected
	X..X	Source cell 1 bit strips mask: 1 = selected; 0 = not selected
2 ... 0	XXX	Multi-grained FM output port address of the source cell

End packet instruction:

3	3		2	2	2	2	2			1	1	1	1	1			1	0	0	0	0	0	0	0	0	0
1	0		8	7	6	5	4	3			8	7	6	5	4			0	9	8	7	6	5	4	3	2	1	0

			Value	Comment
			1	Instruktions-Paket
31	...	30	11	ID: End packet for logical stream releasing
			X	Coarse-/fine-grained releasing: 1 coarse-grained, 0 fine-grained
28			X	Local/global route release process: 1 = local, 0 = global
27	...	23	- ... -	Reserved
	...		XXXX	FM output address within the source cell
18	...	14	- ... -	Reserved
13	...	11	XXX	Source cell 8 bit strips mask: 1 = selected; 0 = not selected
10	...	3	X...X	Source cell 1 bit strips mask: 1 = selected; 0 = not selected
2	...	0	XXX	Multi-grained FM output port address of the source cell

Data packet:

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	0 9	0 8	0 7	0 6	0 5	0 4	0 3	0 2	0 1	0 0

			Value	Comments
31	..	0	0	Data packet
			X...X	Application data

Figures relating to improved way of routing:
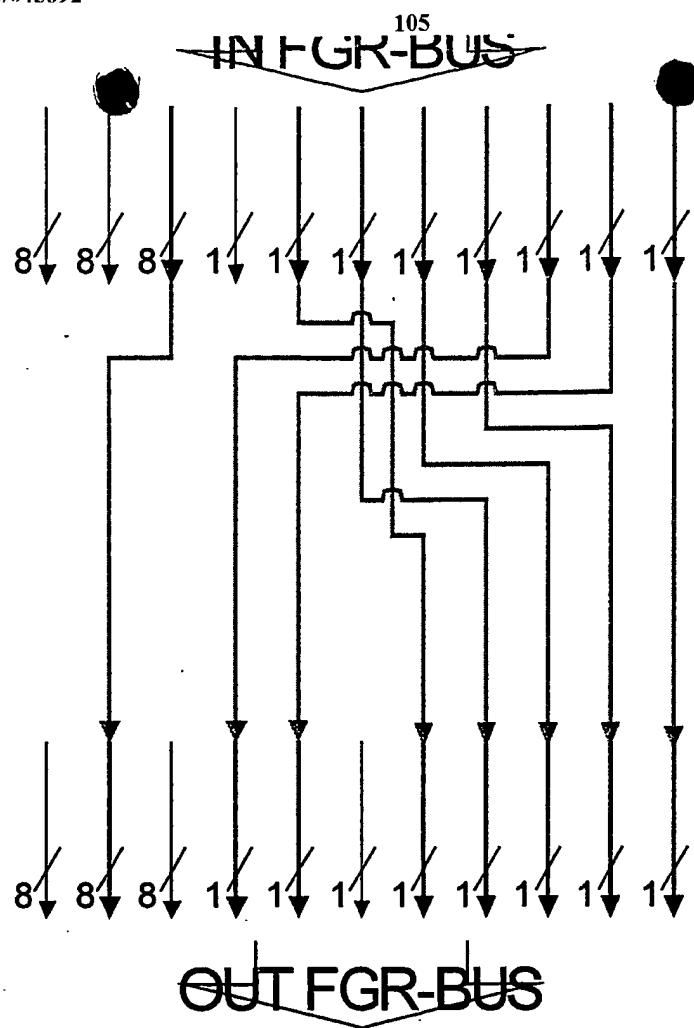
105

IN FGR-BUS

8↓ 8↓ 8↓ 1↓ 1↓ 1↓ 1↓ 1↓ 1↓ 1↓ 1↓

8↓ 8↓ 8↓ 1↓ 1↓ 1↓ 1↓ 1↓ 1↓ 1↓ 1↓

OUT FGR-BUS

→   freier Strip mit 1- bzw. 8-Bit Granularität

→   4-Bit breite logische Verbindung

→   10-Bit breite logische Verbindung

→   1-Bit breite logische Verbindung

→   freier Strip mit 1- bzw. 8-Bit Granularität

(free strip of 1 or 8 bit granularity

4- Bit wide logic interconnection

10- Bit wide logic interconnection

free strip of 1 or 8 bit granularity)

FGR-BUS
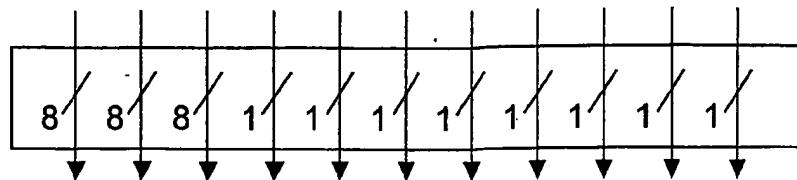
→ 1- bzw. 8-Bit Strips des FGR-Busses

1 or 8 bit strip of FGR-Busses

IN DIR 2, MGR-LINK 1

DIR 3, MGR-LINK 2

→ new incoming mgr stream, in dir 1, mgr-link 1
→ already routed strips, in dir 2, mgr-link 1
→

IN DIR 2, MGR-LINK 1

DIR 3, MGR-LINK 2

→ new incoming mgr stream, in dir 1, mgr-link 1
→ already routed strips, in dir 2, mgr-link 1
→ free available strips

IN DIR 2, MGR-LINK 1

DIR 3, MGR-LINK 2
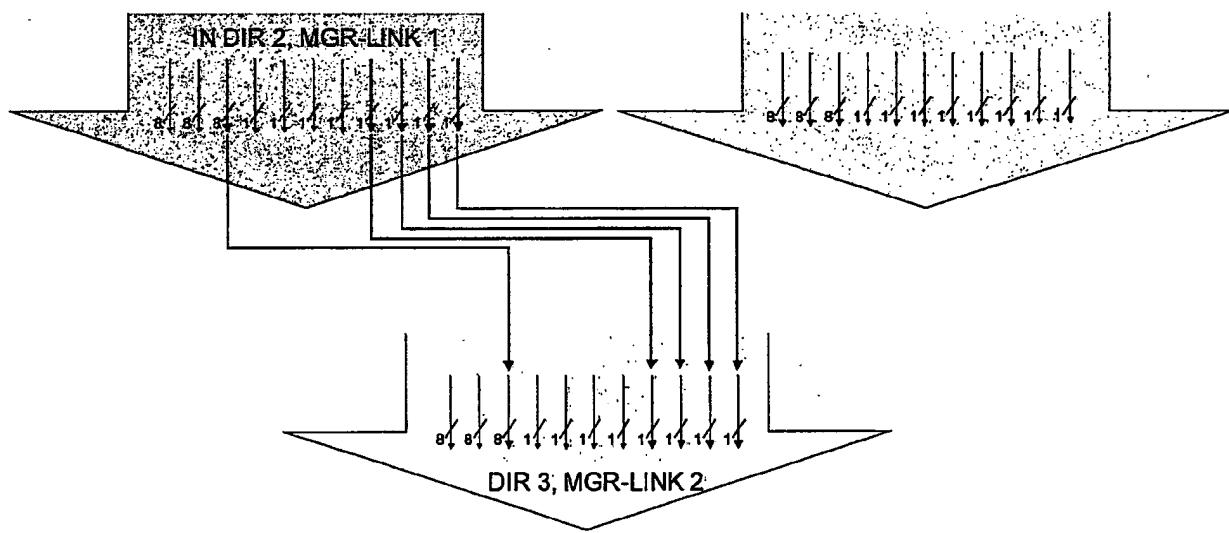
→ new incoming mgr stream, in dir 1, mgr-link 1
→ already routed strips, in dir 2, mgr-link 1

Figures

IN FGR-BUS

8 8 8 1 1 1 1 1 1 1 1

8 8 8 1 1 1 1 1 1 1 1

OUT FGR-BUS
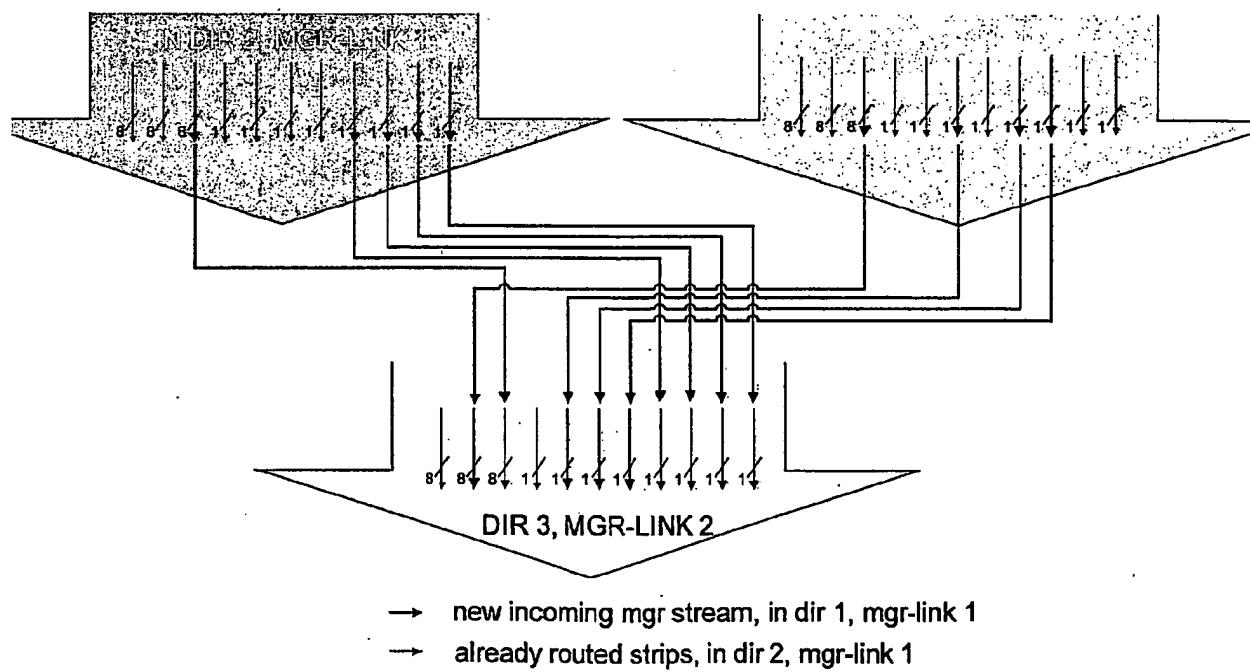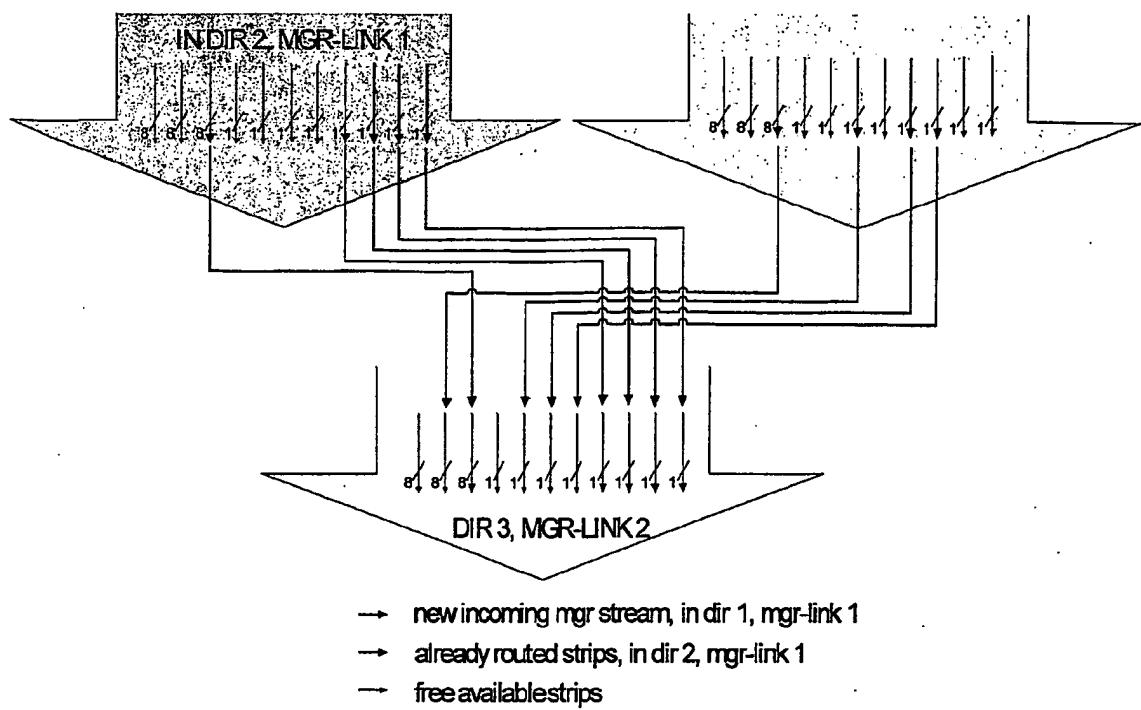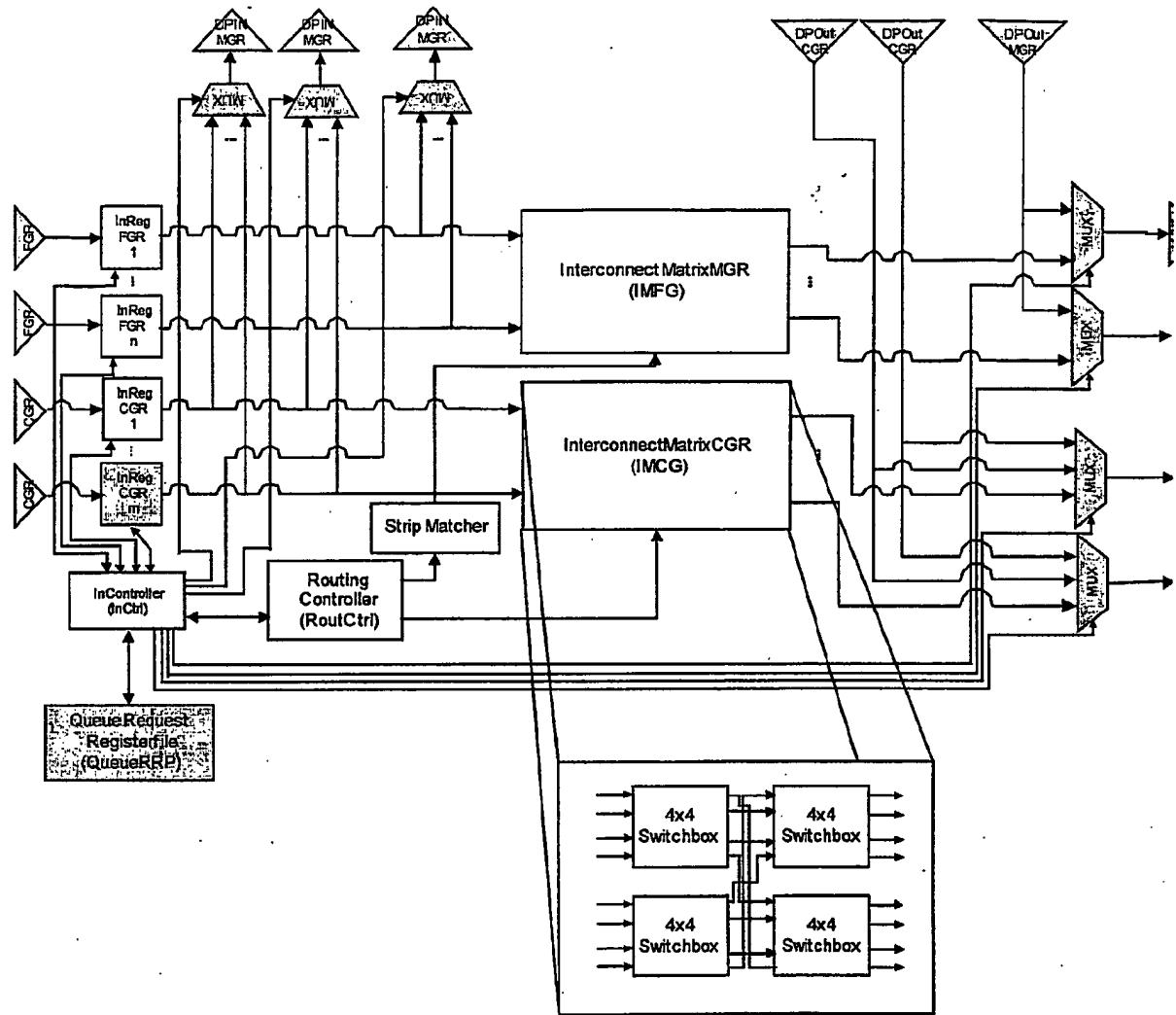
⟶   freier Strip mit 1- bzw. 8-Bit Granularität

⟶   4-Bit breite logische Verbindung

⟶   10-Bit breite logische Verbindung

⟶   1-Bit breite logische Verbindung

⟶   freier Strip mit 1- bzw. 8-Bit Granularität
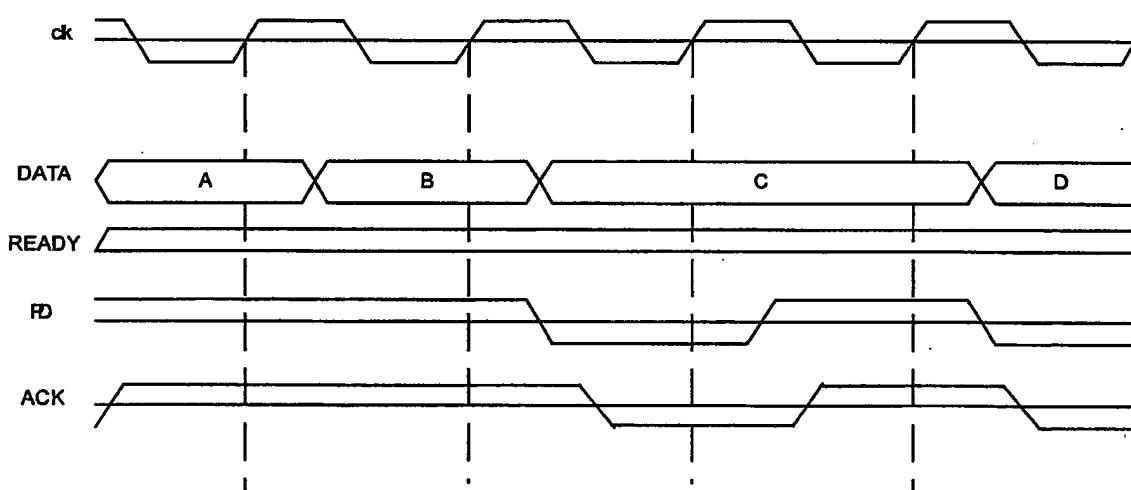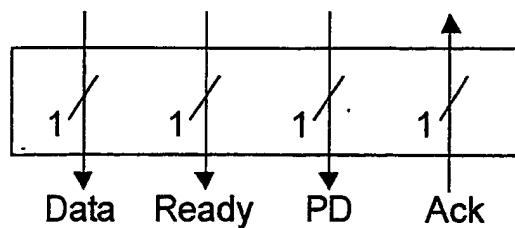
Claims

1. A data processing device comprising a multidimensional
array of coarse grained logic elements (PAEs) processing data
and operating at a first clock rate and communicating with one
another and/or other elements via busses and/or communication
lines operated at a second clock rate,
wherein the first clock rate is higher than the second and
wherein the coarse grained logic elements comprise storage me-
ans for storing data needed to be processed.

2. A data processing device according to claim 1 wherein the
   data processing of the array is controlled in a data-flow-
   like manner.

3. A data processing device according to claim 2, wherein the
   data storage means is adapted for storage of operands and/or
   intermediate results and wherein a valid bit is provided for
   each entry.

4. A data processing device according to a previous claim
   wherein data processing of a coarse grained logic element of
   the array is adapted to be effected in response to all valid
   bits of data and/or triggers needed being valid.

5. A processing array in particular according to a previous
   claim having a main data flow direction, said processing
   array having coarse grained logic elements and said coarse
   grained logic elements being adapted to effect data proces-
   sing while allowing data to flow in said in one direction,

in particular ALUs having an upstream input side and a data
downstream outrput side wherein at least some of said coar-
se grained logic elements have data processing means such
as second ALUs allowing data flow in a reverse direction.

6. A processing array according to the previous claim wherein a
the instruction set for the ALUs in one direction is diffe-
rent from the instruction set of the ALUs in the reverse di-
rection.

7 .A processing array according to tone of the two previous
claims wherein at least one coarse grained logic element
comprises an ALU in one direction and an ALU in the reverse
direction.

8. A processing device wherein the coarse grained element is
connected to the busses and rows of coarse grained ele-
ments are provided interconnected via busses, wherein at
least one input is connected to an upper row and at least
one input is connected to a row below the cell and/or
where this holds for an output connect.

9. A processing device according to the previous claim
wherein the coarse grained element is connected to the
busses and at least two input/output bus connects are
provided in one row and wherein a switch in the bus
structure and/or a gate or buffer or multiplexer is pro-
vided in the segment between inpunt and/or output.

10.   A method of routing a processing array adapted to auto-
matically connect separated fragments of a configuration
and /or configurations and to rip up nonconnectable tra-
ces in a stepwise manner.